

Cheating Prevention in Peer-to-Peer-based Massively Multiuser Virtual Environments

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften

genehmigte Dissertation

von

Sebastian Schuster

aus

Berlin

1.Gutachter: Prof. Dr.-Ing. Torben Weis

2.Gutachter: Prof. Dr. Pedro José Marrón

Tag der mündlichen Prüfung: 31.10.2013

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contribution	3
1.3	Organization	4
2	Massively Multiuser Virtual Environments	5
2.1	Characterization	5
2.1.1	The User Perspective	8
2.1.2	The Operator Perspective	12
2.1.3	Terminology	14
2.2	The Anatomy of a Virtual World	15
2.2.1	World State	15
2.2.2	State Evolution	17
2.3	Client/Server-based MMVEs	18
2.3.1	Locality of Interest	19
2.3.2	Replicated Simulations	19
2.3.3	Player Movement and Actions	20
2.3.4	State Consistency	21
2.4	Peer-to-Peer-based MMVEs	22
2.4.1	Challenges	23
2.4.2	State Distribution	23
2.4.3	Interest Management	27
2.5	Cheating in MMVEs	28
2.5.1	Cheating Types	29
2.5.2	Cheating Countermeasures	34
3	A Cheat-Resistant MMVE Architecture	39
3.1	Basic Idea	40
3.1.1	Event-Based Simulation	40
3.1.2	Optimistic Distributed Simulation	42

3.1.3	Limitations	43
3.2	Architecture	44
3.3	System Model	46
4	A Self-Stabilizing Delaunay Overlay	47
4.1	Requirements	47
4.2	Overlay Structure	48
4.2.1	Key Space and Node Responsibility	48
4.2.2	Node ID Assignment	50
4.2.3	Primary Structure	50
4.2.4	Routing Table	52
4.2.5	Shortcuts	56
4.3	Delaunay Structure Maintenance	57
4.3.1	Self-Stabilization	57
4.3.2	Adding and Removing Neighbours	59
4.3.3	Pending Neighbours and Queuing of NetworkViews	61
4.3.4	Comparing Neighbourhood Views	62
4.3.5	Node Departure	63
4.3.6	Reliability and Resilience	64
4.4	Bootstrapping and Join	67
4.5	Routing Table Maintenance	70
4.5.1	Constraining Routing Table Entries	71
4.5.2	Target Positions	72
4.5.3	Querying Routing Table Entries	73
4.5.4	Key Space Usage	76
4.5.5	Redundant Routing Table Entry Querying	78
4.5.6	Active and Passive Filling	82
4.5.7	Connection Management	84
4.6	Shortcut Maintenance	86
4.7	Message Routing	87
4.7.1	Unicast Messages	88
4.7.2	Areacast Messages	92
4.8	Resilience	97
4.9	Discussion	99
5	A Virtual World Storage	101
5.1	Requirements	101

5.2	Retrieving and Storing Data	102
5.2.1	Data Space and Replication	102
5.2.2	Data Retrieval on Disjoint Paths	103
5.2.3	Storing Data	106
5.3	World State Updating	107
5.3.1	State Distribution and Replication	107
5.3.2	Update Area Retrieval	112
5.3.3	Player Actions	113
5.3.4	Circular Maintenance Area	114
5.3.5	Update Process Timing	115
5.3.6	State Initialization	118
5.4	Storage Maintenance	119
5.4.1	Node Join	119
5.4.2	Node Departure	123
5.4.3	Effects of Churn	125
5.5	Resilience	126
5.5.1	Area Queries	126
5.5.2	Point Queries	130
5.5.3	Preventing Eclipsing	132
5.5.4	Preventing Shortcut Tampering	133
5.6	Discussion	134
5.6.1	Reliability	134
5.6.2	Scalability	135
6	Evaluation	141
6.1	Network Simulation	141
6.1.1	Simulator Environment	142
6.1.2	Event Processing	143
6.1.3	Network Communication	145
6.1.4	Time and Clocks	148
6.1.5	Node Lifecycle	150
6.1.6	Bootstrapping	151
6.2	Common Simulation Parameters	152
6.2.1	Communication Parameters	152
6.2.2	Timing parameters	153
6.2.3	Lifecycle Settings	153
6.2.4	Bootstrapper Settings	154
6.2.5	Overlay Settings	155

6.3	Overlay Evaluation	155
6.3.1	Overlay Test Szenarios	156
6.3.2	Reliability	157
6.3.3	Scalability	170
6.3.4	Resilience	175
6.3.5	Discussion	183
6.4	Storage Evaluation	183
6.4.1	Storage Test Szenarios	184
6.4.2	Reliability	187
6.4.3	Scalability	194
6.4.4	Resilience	210
6.4.5	Discussion	225
7	Conclusions	231
7.1	Summary	231
7.2	Outlook	235
	Bibliography	237
	List of Acronyms	251
	List of Publications	253

Chapter 1

Introduction

1.1 Motivation

Massively multiuser virtual environments (MMVEs) have become an increasingly popular Internet application in the last years. They allow a large number of users to join a simulated virtual online world. There, every user is represented by an avatar. The user controls his avatar to move through the virtual world and interact with his surrounding and the avatars of other users.

The most successful MMVEs are massively multiplayer online role-playing games (MMORPGs). In MMORPGs, users can play a role in a fantasy game world, fighting monsters together with other players, solving epic quests, advancing their characters to higher levels, and acquiring virtual wealth. World of Warcraft [16] by Blizzard Entertainment is the most prominent example, having more than ten million players worldwide as of March 2012 [92]. To play the game, users have to pay a monthly fee, allowing Blizzard Entertainment to make huge profits from operating World of Warcraft. The MMORPG market currently amounts to \$12 billion in 2012 and is expected to grow to \$17.5 billion per year by 2015 [115]. This market size makes creating MMORPGs interesting for game developing companies. Consequently, a lot of new MMORPGs have been announced or already entered the market.

However, there are considerable economic risks involved in developing and operating an MMORPG. After the initial development, there are significant costs to operate the game. Current MMORPGs are based on client/server technology. A central server contains the whole virtual world, receiving player actions from all the clients and supplying them with updates of the world state. This server has to be powerful and it needs a high bandwidth and low latency Internet connection to support just a few thousand players. If a company wants to support more players, it has to use

more servers with each server running an identical copy of the game world, called shard. Operating these servers creates considerable costs.

As long as player numbers are stable and servers are working to capacity, revenues should always cover operating costs. However, player numbers may vary greatly over time and for new games it is pretty hard to predict their success. If companies overestimate player numbers, they will provide too many servers with over-provisioned bandwidth resulting in higher costs and less profit or losses for the company. Underestimating the player numbers also causes problems. When World of Warcraft started, its operator was very much surprised by its success. Servers were overloaded, spoiling the game experience for players with high latency (lag) till the game reacts to player input, overcrowding of areas in game, and even regular server failure. It took the operator quite some time to keep up with the enormous success and open up new shards. At that time, competition in the market was not very intense. However, there are many more competitive games today. Bad playing experience at the start and the resulting bad reputation might ruin the success of a game forever. Users will quit and play one of the competing games.

The need for high player numbers to cover costs of operation and the upfront uncertainty in player numbers form barriers for smaller companies. Without these problems, some niche games appealing to a smaller audience might become economically more successful creating a more diverse market.

Peer-to-peer technology might help to solve these problems. In a peer-to-peer application, there is no central server all clients communicate with. Instead, all peers can communicate with each other, supplying services to other peers and using services other peers offer. Peer-to-peer file sharing applications have been very successful in distributing files, with peers exchanging data directly between each other. No expensive central server with large storage space to store files and high bandwidth to deliver files is necessary. The software creates the network out of participating computers, with each computer supplying storage space and bandwidth to other computers. In an ideal peer-to-peer application, every node would supply as many resources as it consumes.

Thus, a peer-to-peer-based MMVE could possibly support any number of players in one single world, without the operator of the game having to pay for servers or bandwidth. However, to apply peer-to-peer technology in MMVEs, a decentralized simulation of the virtual has to be developed. Instead of using a central server, peers exchange messages directly between each other to perform that distributed simulation. Despite changing network conditions and message latencies, players

expect the simulation to advance as fast as the real wall-clock time, with reactions to player input occurring as fast as they would occur in the real world. Furthermore, the MMVE should work reliably despite nodes joining or even failing unexpectedly. Although several approaches have been discussed, so far no working peer-to-peer-based MMVE exists.

One reason for this is a serious issue threatening MMORPGs: Cheating. MMORPGs are highly competitive environments. Typically, a player advances his avatar to higher levels by fighting monsters and solving quests. He collects better equipment, which in turn enables him to fight even harder monsters and advance further. Virtual currency can be used to buy better equipment, too. Certain achievements can only be reached by investing enough time and having the necessary equipment and player skills. Thus, a player's avatar is perfectly suited to demonstrate the player's status to other players. For a lot of players, gaining achievements and advancing his avatar plays the major role in MMORPGs [128].

How the game is meant to be played is guarded by rules. They govern how players interact with the virtual world and other players and what the effects of interaction will be. These rules have to be the same for all players and they have to be obeyed by all players for the game to be considered fair.

Players playing by the rules have to invest a lot of time and effort to make progress. Consequently, there will always be some players trying to break the rules to get an unfair advantage: to cheat. Since avatars, equipment, and virtual currency are even traded for real money nowadays, there will be motivation and criminal energy for cheating. Cheating poses a serious threat to all MMORPGs. For honest players, fairness is a critical requirement. If honest players think they are competing with cheaters, they will simply quit the game and look for a better one.

1.2 Contribution

In current client/server-based games, enforcing rules is straight-forward. The whole game world and player interaction are centrally managed by a server able to check whether player actions conform to the rules. This server has full authority over the virtual world state and it can be trusted to stick to the rules since it is under control of the operator. In a peer-to-peer-based MMVE, the world state and its evolution is distributed between all peers. These peers cannot be trusted. An attacker could have made any kind of modification to the game program. The best one can hope

for is only a certain number of nodes being cheaters and the majority being honest. Under this assumption, a system using replication and redundant checking might become a viable remedy for cheating.

Our main contribution is the design and evaluation of such a system: a decentralized distributed storage able to ensure the virtual world only evolves to the rules even in the presence of a certain number of malicious nodes. This storage is based on our second contribution: the design of a self-stabilizing peer-to-peer overlay resilient to routing attacks malicious nodes could use to increase their influence on the operation of the storage.

1.3 Organization

This thesis is structured as follows: Chapter two contains the foundations this thesis is based on. It illustrates how current MMVEs look like, explains their current technical infrastructure, and shows what has been done to realize peer-to-peer-based MMVEs. It also discusses the problem of cheating, introduces different types of cheating, and shows the state-of-the-art to counter cheating. Chapter three presents the basic architectural design of our approach. In chapter four, we present a self-stabilizing overlay we developed to serve as the basis of a reliable virtual world storage shown in chapter five. Extensive evaluation follows in chapter six, while chapter seven concludes, summarizing the contributions of this thesis and showing open topics.

Chapter 2

Massively Multiuser Virtual Environments

2.1 Characterization

A massively multiuser virtual environment is a virtual world able to host a massive number of users simultaneously. Richard Bartle gave the following definition of a virtual world in his seminal book [9]:

Virtual worlds are implemented by a computer (or network of computers) that simulates an environment. Some – but not all – the entities in this environment act under the direct control of individual people. Because several such people can affect the same environment simultaneously, the world is said to be shared or multi-user. The environment continues to exist and develop internally (at least to some degree) even when there are no people interacting with it; this means it is persistent.

Multi-user dungeons (MUDs) were the first successful applications to fulfil this virtual world definition. The first MUD was developed in 1978 by Roy Trubshaw and Richard Bartle. MUDs were completely text-based. A user could connect to a MUD server using a Telnet client. He created a player character: an entity representing the user in the virtual world. The state of the virtual world as seen by the player character was presented to the user with text descriptions. It was up to the user's imagination to create an image of the world in his mind. He could issue text commands to control the actions of his player character. He could move to different neighbouring areas, inspect the local area, or interact with other entities in the local area. These commands were sent to the MUD server. It simulated the result of the action, updated the state of the virtual world, and sent the result of the action back

to the user. When entering an area with another player character in it, users could interact by talking to each other or exchanging virtual items. Player characters could also interact with characters controlled by the server, so-called non-player characters (NPCs). Some of them were friendly characters players could talk to and obtain tasks to solve. Others are enemy characters players could fight. Some of these enemy characters continually moved through the virtual world to look for player characters to attack so the world kept changing even when no player characters were present. Thus, MUDs fulfil Bartle's definition of a virtual world.

Bartle pointed out, that the fact that most MUDs were games allowing users to play a role in a fantasy game world heavily influenced the terminology to describe virtual worlds today. Users are *players*. When interacting with the virtual world they *play* the game controlling their *player character*. Although it might be possible to use virtual worlds as e-learning environment or collaborative work environment, most virtual worlds today are games.

While MUDs still exist, modern virtual worlds feature a graphical representation on the client. A user can actually see an image of the virtual world and interaction takes place via a graphical user interface. The first virtual world featuring graphics was Neverwinter Nights published in 1991. It could be played over the AOL online service and existed until 1997. Meridian 59 published in 1996 was the first 3D graphical virtual world. It was also the first to be playable over the Internet. Thus, it can be regarded as the ancestor of modern 3D virtual worlds. Ultima Online published in 1997 and Everquest from 1999 closely followed and were the first to become a commercial success. They still exist today. The most successful virtual world today is World of Warcraft [16] published in 2004, attracting millions of players.

According to Richard Bartle, most virtual worlds can be identified by five characteristics ensuring that a virtual world resembles the real world.

Rules: "The world has underlying, automated rules that enable players to effect changes to it (although not to the rules that grant them this ability). This is the world's physics." Players can execute actions in the world that will change the world. For example, they can move their player character around changing the position of the player character in the world. The result of an action will be computed automatically according to the rules of the virtual world. These rules have been defined by the creator of the virtual world, they cannot be changed by players.

Avatars: "Players represent individuals 'in' the world. They may wield partial or total influence over an army, crew or party, but there is only one game entity that represents them in the world and with which they strongly identify. This is their character. All interaction with the world and other players is channeled through characters." This means a player is represented in the virtual world by his avatar or player character. Actions triggered by the player are executed by his player character. There might be additional characters influenced by the player, but his player character is the most important one. Thus, a multiplayer strategy game is not a virtual world because there is no single character representing the player giving out all the orders.

Real time interaction: "Interaction with the world takes place in real time. When you do something in the world, you can expect feedback almost immediately." Time in the virtual world passes like wall time in the real world. Similar to the real world, actions of characters have almost immediate effect. Furthermore, the world continues to change, even if a player character does not perform any actions. Thus, round-based games would not be a virtual world.

Shared world "The world is shared." Multiple players can join the virtual world at the same time. They can interact with each other directly or indirectly via the environment. A single-player role-playing game would not be a virtual world, as it can only be played by a single player although the first three characteristics might be fulfilled by the game.

Persistence "The world is (at least to some degree) persistent." This means the world maintains its state including the effects of player actions. When a player leaves the virtual world, the changes he affected to the world will remain. When he comes back, his player character will be located at the position he left and the world will have evolved according to its rules and the actions of the remaining players.

While the first graphical virtual worlds were called graphical MUDs, today the term massively multiplayer online role-playing game (MMORPG) is typically used for these games. The term "massively multiplayer" suggests that a high number



Figure 2.1: Representation of the virtual world in World of Warcraft as seen by the player

of players can simultaneously share the virtual world. With current client/server-based MMORPGs up to a few thousand players can join a virtual world at the same time.

2.1.1 The User Perspective

Similar to offline computer games, a player has to obtain the game software and install it on his computer before entering the virtual world. He also has to register at the game operator to create an account. After starting the software and entering account name and password, the user can create a new player character or select one of his existing player characters to enter the virtual world. Typical for role-playing games, a player can choose class and race of his created character and customize its appearance. Most MMORPGs feature a fantasy world and race choices include dwarves, elves, and gnomes. Typical classes are warriors, mages, and priests. A class determines the play style of a character, the available equipment, and the actions that can be performed. Warriors usually wear heavy armour. When fighting, they will try to get close to their target and their actions include various kinds of attacks performed with weapons like swords or axes. Mages wear only thin armour. They will try to keep their distance to the enemy and attack him with magical spells like fireballs. Priests can cast spells to heal characters and are often included when players play in groups to heal injured members. A player can have multiple player characters so when joining he can choose which role he wants to play at that time.

After joining the world, the player will see a 3D graphical representation of the virtual world. Figure 2.1 shows how this representation looks like for the currently most popular MMORPG World of Warcraft [16]. It features a third person perspective with a camera centred on the player character, zoom- and rotatable using the mouse. Player characters can move forward, backward, sideways, and rotate by pressing the associated movement keys. For example, pressing the arrow up key will cause the character to move forward and upon releasing the key the character will stop moving. Similarly, pressing the right arrow key will start rotating right and releasing it will stop the rotation. Holding up and right arrow at the same time will cause the player character to run clockwise in circles.

A graphical user interface is laid over the world view providing additional elements like a map, a chat frame, frames for showing the status of the player character or of the currently targeted character, and buttons giving access to player inventory or player equipment. The map shows a top-down view of the area surrounding the player character. Using the chat frame, players can talk to other players. The character status frames shows the amount of health a character has left and the available resources for performing actions.

Player Actions

Action bars contain the actions that can be triggered by the player by clicking on them or by pressing the shortcut key associated with the slot in the action bar. Actions include attacks or magical spells causing a certain amount of damage to the target character reducing the health of the target. When the health reaches zero, the target dies. Player characters that died will be resurrected. Non-player characters controlled by artificial intelligence (AI) will disappear and spawn again after some time has passed. Healing classes can also cast healing spells to add life points to their own health or to the health of allied characters.

To execute an action, certain preconditions have to be satisfied. Many actions need a target the action is performed on. The player has to select a target from the world using the mouse. It will appear in the target frame and all actions requiring a target will be performed on that target. When the player tries to attack the target, his player character must face the target and it must be within range of the attack. Maximum action ranges differ between actions. Melee attacks typically require being very close to the target while magical spells can be cast from longer ranges. Actions often cost some kind of resource such as mages needing mana to cast a spell. Only when enough of the resource is currently available, the action can

be performed subtracting the cost of the action from the available resource. Often resources regenerate over time. If the player triggers an action and the preconditions are not fulfilled, he will receive an error message.

Otherwise, the action will be executed by applying effects to the world. A typical effect is causing damage to a character reducing the amount of health the character has left. The damage caused depends on attributes like strength, stamina, and intelligence of the player character as well as attributes of the receiving character like armour or damage resistances. These attributes depend on the level of the player character and its equipment. Player characters with higher level and better equipment are stronger, able to defeat stronger opponents.

Actions like magical spells often have an associated cast time that must pass before the effect is applied. For example, casting a fireball may require the player character to stand still and wait for two seconds before the fireball is actually launched and flies towards the enemy causing damage on impact. Any movement of the player character while casting will interrupt the action. Enemies getting in melee range of the caster might also perform a special action interrupting the casting. Upon execution of the action, the action must typically cool down for a certain time until it may be used again. Cooldown times are different for different actions with more powerful action having longer cooldowns so they can be used less often during a fight. There is also often a global cooldown that prevents any action from being executed after a former action has been triggered. In World of Warcraft, this global cooldown lasts 1.5 seconds but can be reduced by a certain attribute of the player character. This mechanism fosters a tactical use of abilities activating the right ability at the right time instead of letting the player hitting keys fastest win. The global cooldown further decreases action frequency to limit the amount of actions the server has to process in a given time.

Considering the large variety of actions of different classes that also includes healing, temporary strengthening effects for allies or weakening effects for enemies, and effects hindering or fastening movement of characters, fights feature interesting dynamic interactions. The interaction is usually not as fast-paced as in first-person shooters (FPS), but more tactical due to cooldowns and the wide variety of actions. The winner in player versus player fights (PVP) is determined by player skill as well as strength of player characters. NPCs can also perform certain actions but their AI is more predictable than a human enemies' actions making defeating NPCs mostly a matter of player character level and equipment.

How and how fast characters move, the set of available actions, their preconditions, effects, and how these effects are calculated make up a major part of the game mechanics or game rules. It is absolutely crucial these game rules cannot be broken as we will argue in the next section.

Motivations of Play

Fighting NPCs or other players makes up a large part of player activities but is just a means to an end. The world in role-playing games has a story players can experience. NPCs will send players on quests, asking them to defeat certain monsters in the world, to protect NPCs from enemies, or to collect certain items that are hidden and protected by monsters in a distant location. Sometimes multiple quests are chained together to tell a bigger story. By solving quests and defeating NPCs, player characters receive experience points. Upon receiving enough experience points the player character will advance to the next level, raising its attributes making it stronger. Quest rewards also include improved equipment also increasing player character's strength. After becoming stronger, the player character can go to other areas of the world with stronger NPCs and solve quests and experience the story there.

In contrast to offline role-playing games, MMORPGs allow to play together in a group to solve quests and fight monsters together. This group play is often encouraged as defeating the strongest NPCs requires coordinated play by a group of players fulfilling special roles like "tanks" absorbing damage, healers healing wounded players, and damage dealers bringing the enemy monsters down.

Virtual worlds also have an in-game economy where goods can be crafted from reagents in the world and items can be traded between players using an in-game currency. The primary purpose of obtaining these items is again to become stronger. However, some players also enjoy getting rich by crafting, buying, and selling items.

Richard Bartle developed a typology of players classifying them according to their motivation for playing an MMORPG [9]. According to him, there are achievers, explorers, socializers, and killers. Achievers primarily focus on advancing their player characters, becoming more powerful, richer, or obtaining the best equipment. Explorers concentrate on exploring the world, experiencing the quests and the entire content that is available. Socializers primarily enjoy playing together with friends and the social interaction among players. Finally, killers enjoy playing the role of a bad guy killing other players and disrupting their game experience.

Yee [128] argues this taxonomy of types is too strict as there could be multiple overlapping motivating factors. He conducted a survey among 3000 players asking them for their motivation. Afterwards he classified the motivation into the three main factors achievement, social, and immersion. They were divided into several sub-factors. For example, the achievement factor is divided further into advancement, mechanics, and competition. Players focusing on advancement again enjoy increasing level and strength of their player characters to show off their status to others. Players focusing on mechanics have fun analysing the game rules to maximize their strength becoming the most skilful players. Finally, players who like competition especially enjoy defeating other players in fights or dominating the economy of the world.

Advancement and competition were among the most important factors mentioned by players. If competition and showing off the obtained status is that important to many players, it is absolutely crucial the competition is fair. The rules should be the same for all players. If some players are able to cheat and break the rules, honest players enjoying the competition will simply quit playing the game as they cannot win an unfair competition. This endangers the success of the virtual world. Thus, operators have to make sure cheating is not possible.

2.1.2 The Operator Perspective

Developing and running a virtual world causes considerable costs for the operator. As of 2003, the average costs for development were estimated to be \$7 million with \$10 to \$12 million being common [95]. Lately, the development cost of the MMORPG *Star Wars: The Old Republic* [39] were estimated to be the highest ever totalling \$150-\$200 million [96]. The main reasons for these high costs compared to developing traditional offline games are twofold.

First, a huge amount of content has to be created for the virtual world. For offline games, it is appropriate to offer 20 hours of entertainment until the game is played through. MMORPGs are designed to be played continuously. Every time a player enters the world, he needs something to accomplish. With many players in the world, the world has to be big to accommodate all the players and provide a variety of activities to choose from. Creating content like quests and 3D models for world objects requires considerably bigger effort compared to offline games.

Second, designing and implementing the technical architecture for an MMORPGs is much more challenging. Currently, it consists of servers running the virtual world

simulation and clients visualizing the world and allowing player interaction. Even with this centralized architecture, an MMORPG is a distributed system and developers have to face all the additional difficulties like partial failure and message loss. Still, the virtual world should always be available, run reliably, and it should be scalable so game interaction runs smooth even with a massive number of players. If player progress is lost, the world is not available, or interaction suffers from message delay or inconsistencies, players will be disappointed and quit eventually.

In addition to the initial development costs, launching the games requires investments into server infrastructure and marketing. Afterwards, there are costs for running the virtual world, estimated to be as high as \$3-\$5 million per year [95]. These include further developing and maintaining the world and adding new content but also staff costs for player support and community management. Finally, the energy and bandwidth consumed by servers and any additional maintenance costs have to be paid as well.

A commercial operator of a virtual world expects revenues to make up for the running costs, to pay off the initial investments, and to generate a profit. The classical subscription-based model requires the player to first buy the game for a price comparable to offline games, covering a considerable part of the development costs but not generating a profit on its own [95]. Typically, 30 days of play time is already included. Afterwards, players have to subscribe by paying a fee to continue playing. This fee usually ranges from \$10 to \$15 per month, depending on the game and the subscription interval. World of Warcraft employs this subscription model generating a large stream of revenues with its 10 million subscribers as of 2012.

Lately, the so-called free-to-play model has been applied successfully by MMORPGs like Lord of the Rings Online [116]. With this model, downloading the game client, creating an account, and playing the game is free of charge. However, after the player started playing and advanced its character some levels, the game will begin to charge the player for making additional content available. For example, higher-level regions of the world will not contain any quest givers and unlocking them has to be paid for first.

There is an in-game shop where players can buy these quest packs or a wide variety of items to increase the advancement speed or make playing the game more convenient. More character slots to create more player characters than the initially allowed number can also be bought. The operator assumes once a player started playing the game and got attracted to it, he will pay to continue playing. Even if he does not, he is still available to play in groups or build social connections with others

who in turn pay to play. Considering the fact that Lord of the Rings Online runs on this model for quite some time and a lot of other MMORPGs have changed to this model lately, it seems to be quite successful.

The third model called pay-to-play is a hybrid between the two former models, requiring the player to pay for the game initially but no subscription is necessary. Guild Wars [3] and Guild Wars 2 [4] are examples for this kind of model. Guild Wars 2 also features an in-game shop generating running revenues.

No matter what model is used, having more players will generate more revenues. The major part of the costs is fixed like the development costs or staff costs for game maintenance and support. It does not make difference in cost whether a development team enhances the world to keep it interesting for 100.000 or 1 million players as they create the same content for every player. Some staff costs like support or community management might vary with player numbers. Bandwidth costs are probably the only costs that directly scale with the number of players. Therefore, a commercial game operator will always try to attract as many players as possible. When fixed costs are covered, every additional player generates a profit margin. Thus, operators are naturally interested in keeping the virtual world cheat-free so players stay attracted to it.

2.1.3 Terminology

A lot of terms have been used to describe what is essentially a virtual world: distributed virtual environment (DVE), networked virtual environment (NVE), massively multiuser virtual environment (MMVE), massively multiuser online game (MMOG), or massively multiuser online role-playing game (MMORPG). Each of these terms stress different aspects, for example DVEs and NVEs are used to describe systems with smaller user numbers while all M*-terms include these massive user numbers. The term MMVE we use is more neutral towards potential uses of virtual worlds, also allowing systems like collaborative workspaces. In fact, MMORPGs like World of Warcraft with their 3D representation and their fast-paced fighting interaction in a fantasy game world are the main motivating and as of today only existing applications. Therefore, a user is a player and we will use the terms MMVE and MMORPG quite interchangeably.

There is a clear distinction between the terms player, player character, and node or peer of a player. A player is the user in front of the computer controlling the actions of his player character in the virtual world. A node or peer is the process on the

player's computer part of our peer-to-peer network. However, to improve readability we will often use the term player instead of player character when the true meaning is clear from the context. We will also talk about the current position of a player or even the current position of a node when we are really referring to the current position of the player character of the player running the node. Where necessary and terms could be mixed up we will draw distinctions and use the original terms.

Finally, when we say a player performs an action we really mean a player makes its player character perform an action. In a strict sense, this action is an action allowed by the game rules but not including movements. However, sometimes we will use the term action including both movement and actions in the strict sense to denote everything causing a change of world state triggered by a player.

2.2 The Anatomy of a Virtual World

Internally, a virtual world consists of a set of logical objects, also called entities. Each of these objects has attributes describing its state. The state of all objects in the world determines the state of the world. This state evolves over time as the world is simulated according to the world rules. Multiple physical copies or replicas might exist for each logical object on different hosts. In fact, every host rendering a view of the world needs a replica of the objects it renders or at least a partial copy of the object state to create a visual representation.

2.2.1 World State

The primary attribute associated with every object is its position. It decides where in the virtual world the object is located. Positions are defined using a three-dimensional coordinate system based on floating point numbers. Most virtual worlds are similar to the real world in that there is a ground plane with x- and z- coordinate defining the position on the plane. The y-coordinate specifies the height. The world is bounded with maximum and minimum values for x-, y-, and z-coordinate defining a cuboid space for the world. It mainly extends into the x-z-plane. Like the real world, virtual worlds are mostly flat. While it would be possible to use a spherical or toroidal shape for the world, it is very uncommon because it is more intrusive when new areas are added to the world. Thus, worlds have borders and new areas can be added behind these borders.

Typically, every object has got an associated ID uniquely identifying the object among all others in the world. Furthermore, each object has got an associated graphical 3D representation realized using a textured 3D model to be rendered by the game client. The orientation attribute determines which direction in the world coordinate system the 3D model is currently facing.

Depending on its type, an object may have various additional attributes. Some of these may change dynamically over time, like the position and the orientation of an object if it is mobile. Others may be static like the ID of an object. Objects that have only static attributes are called static objects. The vast majority of the world is made up of static objects. The complete exterior of the world, the ground, mountains, buildings, and trees down to beds and chairs within houses are all static objects. They will never change so it is not possible to dig a hole in the ground, to chop a tree, or to move a chair.

Dynamic objects with variable attributes includes things like doors that can be opened and closed or chests that can be found in caves revealing valuable items when opened. These objects are typically immobile but player characters can interact with them changing their state. Mobile objects have additional attributes describing their current movement: a 3D vector describing direction and the speed of movement and another vector to describe the rotation of an object.

Most mobile objects are characters, moving through the world. This includes AI-controlled non-player characters like animals and monsters players can fight with or human characters players can talk to and obtain quests. These characters have attributes describing the powers they have, their current health, and the current state of the AI containing its plan or the goal it wants to accomplish. A player character has a lot more attributes, also describing its equipment, the items in its inventory, current action cooldowns, or the progress on different quests.

Objects like chairs might actually be dynamic and moveable in single-player offline games. In a shared world, the opportunities for players to change the world are purposefully limited. If chairs could be moved in an MMORPG, there would definitely be players who would take them to inappropriate places like the dragon's lair disrupting the game experience of others. Therefore, changes to the world are mostly limited to fighting and killing monsters in MMORPGs. The monsters will reappear after a while. This way the world always returns to the same state so all players can have the same experience.

The limitation of player influence on the world is also the reason why the inventory and the equipment of player characters are modelled as attributes of player charac-

ters instead of objects in the world. If items can be placed anywhere in the world, players could easily disrupt other players' game experience.

2.2.2 State Evolution

A virtual world resembles the real world. Time advances like wall time in the real world and the state of the world changes continuously. Upon triggering an action, players expect an immediate response. This continuous real-time behaviour has to be simulated by a discrete step computation device. Therefore, discrete updates of the world state are calculated at a very fast rate, typically 60 Hz. This rate is high enough for humans to only perceive a continuous change, creating the desired impression.

Internally, two functions are used to realize this: *Update()* and *Draw()*. *Update()* calculates the new state of the virtual world and afterwards *Draw()* renders the part of the world visible to the player. At a frequency of 60 Hz, both functions will be called every 16.66 ms. Typically, the operating system is not a real-time system so invoking these functions at precisely the intended time cannot be guaranteed. However, on typical powerful systems 60 Hz should be reached on average. To prevent jitter in the update frequency from causing stuttering in the world, the time that has actually passed since the last update is also given as a parameter to the update function.

Update() inspects all dynamic objects in the world and calculates their updated state based on the old state and the amount of time passed since the last invocation of the update function. If Δt is the time since the last update, then the new positions p_{new} of mobile object can be calculated using the formula $p_{new} = p_{old} + \vec{v} * \Delta t$ where \vec{v} is the current movement vector of the object and p_{old} is its old position. Similarly, the orientation of the object can be updated using its rotation vector and the passed time.

To be more realistic, the position updates are usually done using more sophisticated physics calculations. For example, while moving on the ground, the height of the objects is always set to the ground. If a character has an upward moving vector because it is jumping, gravity will change its movement vector over several updates so it will return to the ground. Friction can be simulated causing characters to slide down inclined planes so characters cannot walk up steep mountains. Movement of characters is also stopped if their 3D model starts to overlap with other objects like buildings or other characters so a player cannot just walk through walls.

During the update, the AI state of NPCs is also updated, e.g. inspecting the surroundings of the NPCs and looking for close player characters to chase and attack. The update function also processes player input by sampling the state of player input devices. If the forward key is detected as being pressed when it was not pressed on the last update, the movement vector of the player character is changed to moving forward. Similarly, if it was pressed but is detected as released, the movement vector component of the forward direction will be changed to zero. If the player presses a shortcut key to trigger an action, the update function will check whether all preconditions are met and start executing the action. It will add the action to the set of running actions. On further invocations, it will advance the state of any action in this set. It will subtract the elapsed time from the remaining cast time and apply its effects by modifying the state of the corresponding objects. When the action is finished, it will be removed from the set.

Movement of characters and most actions also have associated animations so players can actually see what is being done. Starting and updating the state of animations and blending different animations into another also happen inside the update function. When the update is finished, the draw function renders the current world state. The 3D rendering is usually computationally much more expensive than updating and will take much more time than the update. When drawing has finished, any remaining time is skipped until the update-draw-cycle starts again. This basic principle is also used in distributed multi-user virtual worlds, as we will show in the following sections.

2.3 Client/Server-based MMVEs

Today's MMORPGs like World of Warcraft [16] are all based on client/server technology. The game provider operates powerful servers in a data centre simulating the virtual world or parts of it. With millions of subscribers, no server would be able to run a virtual world with that many players. Therefore, MMORPGs use a concept called *sharding*. Instead of one single virtual world, there are multiple shards each representing another copy of the virtual world. When creating a player character and joining a world, the player has to decide which shard he wants to play on. Each shard is run by its own servers. The player connects to the game server of its shard using his game client. The server sends the avatars of the player and the player selects one to join the world. Then the server sends the current state of objects to the client that will render the scenery around the player character.

2.3.1 Locality of Interest

Sending all objects in the world to all players would not be scalable. Therefore, the amount of transferred data is reduced by sending only the state of dynamic objects to the client. Static objects which make up the majority of objects are already hard-coded into the client. Second, only the objects close enough to the player character for him to see and interact with are actually sent. This limited visibility and interaction range creates a circular so-called area of interest (AOI). Its radius is commonly smaller than the real-world visibility with typical values around 50 meters.

This *locality of interest* is the reason why distributed virtual environments can actually scale to massive user numbers. If all clients were always interested in the whole world state, the amount of data sent to each of the clients would increase with every additional player quickly overloading the server. As long as players are evenly distributed all over the world, this locality limits the amount of data the server has to transfer. However, if there are hotspots in the world where a lot of players aggregate so they are all in each others' AOIs, the server might still become overloaded. In the worst case, all player characters are in the same location and the system would not scale again.

2.3.2 Replicated Simulations

The server continuously updates the state of the virtual world as described in section 2.2.2. Since the server knows the positions of all player characters, it can send every client only the updated state within its AOI. Sending every player the state of its AOI 60 times per second would overload any server. Therefore, the client performs its own simulation of the local area. As long as no new objects enter its AOI or other player characters in its AOI perform actions, the simulation on the client will update the state on its own trying to make it consistent with the state on the server. However, standard consistency models like linear, serial, or causal consistency [60] cannot be readily applied as they do not fit well with the continuous changes of state, causing high response times [55]. Therefore, objects will only be loosely consistent as shown in the following.

When a player starts to move, its client will send a message to the server and it will include its movement into the local simulation beginning to move the player character. Upon reception of the message, the server will also begin to include the movement of the player character and it will calculate the set of clients whose player

characters are in visibility range to the now-moving player character. It will send each of these clients a message so they can also include the movement of the player character in their simulation.

However, transferring messages takes a certain amount of time. When a player character starts to move, it will start moving immediately on its client. When the message arrives at the server it will start moving there, and when the messages from the server reaches the clients whose AOI the player character is in, it will start moving at these clients. Since the messages for the different clients can arrive at different times, the movement will not necessarily start at the same time on all clients. All clients and the server will see a different state with the moving player character being at a slightly different position depending on when the movement was started. These inconsistencies can easily be observed when two players are physically next to each other and start running forward at the same time as every player will see himself being in front. These small inconsistencies can grow fast considering that starting and stopping to move in conjunction with turning at slightly different times can lead to positions finally diverging a lot.

Therefore, the server is the only authority on the current state of the world and it can overwrite the states created by the simulations on the clients with its own. This way it prevents inconsistencies on the clients from becoming too big so the resulting inconsistency is a short-term inconsistency [89]. However, simply overwriting the state at the client could be very disruptive as positions of objects suddenly change when the client receives the updated state. Therefore, clients usually interpolate between their own state and the received state to create a smooth transition to the new state. This overwriting of the state mostly happens indirectly on the fly, as all player or NPC actions sent to the clients also contain the current position of the acting characters. This way the state never diverges too much.

Often there are additional dynamic objects in the world that are not relevant to any actions performed in the world but they make the world appear livelier. This includes birds flying in the sky or grass on the ground swaying in the wind. The dynamic states of these objects are never synchronized between clients and server as they do not exist on the server in the first place.

2.3.3 Player Movement and Actions

As player characters and their AOI move, the server will continuously send them the current state of objects entering their AOI. Similarly, if objects move into the

AOI of a stationary player characters the client will be informed about their state by the server so it can include the objects into the local simulation.

When a player triggers an action, the client will locally check whether the preconditions for executing this action are met. If they are met, it will start executing the action and send a message to the server. The server will perform the same check and either start the action or respond with an error message. This can always happen as the state of client and server is not necessarily consistent. An enemy player could be just within range of a certain attack while the server already noticed that player started moving and is already out of range. The client will stop any animation it started for the action and display the server's error message saying the target is out of range instead. The higher the latency to the server, the more likely inconsistencies and disruptive overruling by the server occur.

The effects of any actions are also calculated by the server. While the client may already start an animation for an attack, the final damage caused, often involving some randomness, is only decided by the server. The same is true for actions performed by NPCs. The client might simulate and perform all actions but the final effects on object states can always be overridden by the server. Animations played for actions are not synchronized with the server as the server does not include animations in its simulation. Animations serve the same purpose as the nonrelevant objects to make the virtual world appear more realistic but they do not play a role in any state update calculations. Therefore, the times an attack animation hits an enemy character is not necessarily the time the server subtracts the damage from the health of the target.

2.3.4 State Consistency

In effect, the world state on clients and server is only loosely consistent [56]. However, players could only see this if they did a side-by-side comparison of the views presented by multiple clients. Besides that, players might only notice this indirectly when positions of objects suddenly change or actions that should have been possible from the client's view are denied because the client state was corrected by the server. These inconsistencies can still be disruptive or even lead to players taking wrong decisions based on the wrong state.

As long as players take actions and messages are in transit, the state of the world will be different on server and clients. The world state will be eventually consistent [120] so all clients and server will have a consistent state if they are not moving and

not performing any actions. However, it is hard to define consistency for systems with continuous state changes. Traditionally, replicas of an object are said to be consistent at time t if they have the same state. This definition has been adapted by Mauve [89] for use in continuous systems defining the state to be consistent if all hosts that received all messages necessary to calculate the correct state at time t have the same state at that time. Even this adapted definition fails to really capture the nature of continuous systems.

The times at which different hosts calculate their updates are never perfectly synchronized. When an object is moving and multiple hosts update its position, they would have to do this at exactly the same time to calculate exactly the same position. Even if all of them update the state of the objects within one millisecond, they will still generate slightly different position and the replicas of the object would not be consistent according to the former definition. On the other hand, if all the deviations are only caused because of the small differences in update time, this would still be sufficiently consistent.

Therefore, our definition of consistency at time t defines world states to be consistent if updating the state from the last updated state to time t on all hosts that received all messages necessary to calculate the correct state at t yields the same state. Hosts with clocks running ahead of the true time will already have updated their local state to time $t + x$. Their state is considered to be consistent with the state of others if it is reachable from the other's state at time t .

2.4 Peer-to-Peer-based MMVEs

First proposals to implement distributed virtual environments on decentralized architectures reach back to the 1990's targeting multiple users and distributed architectures e.g. SimNet [20] or MiMaze [49]. These architectures could only support small user numbers since they used a decentralized architecture broadcasting all state updates to all nodes. This was shortly followed by rising popularity of the first successful MMORPG Ultima Online [38].

Later, peer-to-peer file sharing networks like Napster and Gnutella [57] became extremely popular allowing a large number of users to share files without needing expensive central servers and having to pay for file transfer traffic. Therefore, interest was rising to implement MMVEs based on peer-to-peer technology, freeing operators from the costs of running expensive servers and paying for their bandwidth. Instead,

the bandwidth of clients/peers players already paid for is used. Furthermore, if designed right, peer-to-peer networks can scale very well. Applied to MMVEs, much larger virtual worlds could be realized and sharding would not be necessary.

2.4.1 Challenges

In a peer-to-peer based MMVE, there is no central server holding all objects in the world, updating them based on the actions received from clients, and distributing the updated state to interested clients. The objects in the world have to be distributed among the peers as no single peer is able to receive actions of all players and update the world state.

Consequently, events like the actions and movement of players have to be forwarded to peers responsible for the area the players are in. They are the authority for the state in that area. Players moving into this area will receive the state from these nodes. Furthermore, nodes in an area need to see the actions and movements of other player characters in that area so they can update their local simulation. The mechanism to get updates and actions to peers interested in these events is typically called interest management.

The peers have to form a structured overlay network on top of the underlying TCP/IP layer, to find peers based on their locations and exchange messages efficiently. Furthermore, redundancy has to be employed to compensate for the unreliability of peers so the world is simulated correctly despite nodes joining and leaving unexpectedly. In the following, we will analyse how existing approaches try to realize peer-to-peer based MMVEs focusing on their state distribution and their interest management mechanism.

2.4.2 State Distribution

After unstructured peer-to-peer networks became very popular, structured peer-to-peer networks realizing a distributed hash table (DHT) like Chord [113] and Pastry [103] to store and look up data in a purely decentralized fashion have been developed. In DHTs, the connection graph is a special structure, e.g. a ring. Nodes in the ring have identifiers from a one-dimensional numeric key space and they are responsible for storing all data from a certain part of this key space, e.g. all keys closest to their own identifier. When storing or looking up data, a node can always

decide whether it is responsible for a key or whether any of its neighbours in the structure is closer, forwarding the query to the closer neighbour.

In [70], DHTs were used as a backup persistent storage. Since DHTs generate keys for data using hash functions, they destroy any locality of data as they map data uniformly to the key space. Objects close in the world will have keys far apart so they will not be stored on the same node in the network. This makes executing range queries to retrieve state in an area very expensive.

Therefore, Mercury [15] used the basic ideas of distributed hash tables to form a structured overlay without using hash functions. It directly used the attributes of stored objects to map objects to the nodes responsible for storing it arranged in a ring structure. Thus, objects close in the attribute space were also close in the ring and range queries can be executed more efficiently. Mercury used multiple ring structures to support multiple attributes. In Colyseus [14], Mercury was used to store objects from a virtual world with 2-dimensional positions by using x- and z-coordinate of the positions as object attributes. However, having to route queries on multiple hops before data can be retrieved proved to be too slow considering the real-time interaction necessary for virtual worlds so Colyseus was only used for smaller multi-player games. Similarly, [37] proposed to use a spatial index on top of a generic DHT to store objects based on their locations. Again, maintaining the index and multi-hop routing in the DHT will be too slow for real-time interaction.

Therefore, other approaches combined ideas of peer-to-peer-based and client/server-based MMVEs. Typical for these approaches is the splitting of the world area into zones and using coordinators responsible for data in their zone. As the world is mostly flat and objects are mainly distributed in the x-z-plane, the splitting typically happens using these two dimensions. This is common to all approaches distributing world state, including ours.

In [70], the world is split into disjoint fixed size zones. In each zone, there is one node responsible for updating the state in that zone essentially acting exactly like a central server for that zone with nodes forming a zoned federation. An underlying DHT is used to assign responsibilities for zones and serve as a backup storage. However, seamless change of zones was not possible because each zone was like a small virtual world with its own central server.

Therefore, [6] split the world into fixed size rectangular zones with one coordinator per zone, allowing players seamless changing of zones. Whenever a player gets close to a zone border and his actions may affect multiple zones, coordinators use locking mechanisms to synchronize state changes. However, distributed locking

only works if nodes holding locks are reliable. This system is targeted more at using multiple reliable servers than unreliable peers. Hydra [24] follows a similar approach, sacrificing seamless zone changes but implementing failover mechanisms for servers using replicated backup zones on different servers.

Mopar [129] partitioned the world into small hexagonal cells also allowing seamless transition between cells. Each cell has an associated home node which is determined using a Pastry DHT to map cell identifiers to the responsible nodes. This home node always knows which node currently located in a cell is the master node of that cell acting as its coordinator. All other nodes in a cell are slave nodes connected to the coordinator using it to exchange events and update their world states. Master nodes form an overlay connecting to all neighbouring master nodes, each updating the state in its own cell and synchronizing object transitions with its neighbours. Furthermore, they exchange information on player movement changing master node responsibilities and recording changes in master nodes at the home nodes so other nodes can always discover the master node of a cell using the DHT.

SimMud [80] uses a combination of a Pastry DHT and coordinators to simulate the virtual world. Similar to [70], the world is split into disjoint fixed size zones with no seamless transition. For each zone, there is a multicast group using Scribe [22] on top of Pastry to realize application-level multicast. Every node periodically reports its player character's position and any state updates to all other nodes in the group. The other nodes use interpolation and dead-reckoning [110] to smoothen the visual representation of player character movement. Furthermore, each group has a coordinator responsible for updating the state of all non-player characters based on the actions of player characters and sending state updates of NPCs to its group. However, using Pastry's multihop-routing to realize a scalable multicast with Scribe incurs a considerable delay for propagating updates and was too slow for real-time interaction.

All of the coordinator-based approaches so far use fixed size regions. The load on each coordinator depends on the number of players in a region correlated with its size. However, some areas might be more popular than other areas creating a higher load on coordinators responsible for these areas. Even approaches supporting different region sizes [24, 70, 80] require the developer to estimate popularity of regions beforehand and they cannot adapt to changing popularity over time. Therefore, coordinator-based approaches have been proposed using a dynamic partitioning of the world into zones.

In [101], peer-to-peer technology is used to manage the server infrastructure simulating the world partitions. Similar to former approaches, the servers act as coordinators for their respective zone. Zones are dynamically generated similar to the 2-dimensional DHT CAN [98]. In CAN, every node is responsible for a rectangular zone and nodes in neighbouring rectangular zones are connected. Messages can be sent using the location as address, arriving at the node responsible for the zone the location is in. With one peer in the network, its zone would be the whole world. When another peer joins, it will send a message to its position arriving at the node responsible for the respective zone. That node will split its zone in two equally-sized zones and each of two nodes will become responsible for one of the split zones. In [101], servers form a CAN overlay and peers connect to the servers based on the zones they are currently in. These servers simulate the world state in their zone. They are connected to the servers of neighbouring zones so they can handle object transition and synchronize state. If a server experiences high load because too many players are in its zone, a new server will be added to that zone halving its size and reducing the load on that server. Similarly, servers with low load can be removed joining zones that have been split. This allows better load balancing than possible with fixed size zones. The hybrid architecture HYMS [28] also uses the CAN-based recursive splitting approach to control responsibility of servers for zones.

GROUP [19] uses a CAN-based overlay to store the world state in a completely distributed fashion. In GROUP, the peers themselves are assigned fixed random positions in the world and are responsible for storing and updating all objects within their CAN-zone. These zones change dynamically as node join and leave the network and the objects have to be passed around. This dynamics was taken one step further with approaches using the current positions of player characters in the world to determine the responsibility for objects in the world. When every node is responsible for storing all objects it is closest to, a Voronoi tessellation [7] is created where each peer is responsible for all objects within its Voronoi cell, see also figure 4.1. Distributing state based on Voronoi tessellations is used in several approaches like [13, 18, 67, 99]. They all have the advantage of a more fine-grained load balancing as nodes join the network or player characters aggregate in an area automatically shrinking zones. However, they also suffer from the increased overhead associated with transferring objects between nodes whenever player characters move. Therefore, using dynamic Voronoi tessellations might be more suitable to disseminating events as shown in the next section.

2.4.3 Interest Management

Interest management mechanisms create an overlay structure allowing efficient dissemination of events to all nodes interested in these events. These nodes can update their player character's state if affected by an event and they can update their current world view. The interest management also has to adapt the structure of the overlay as players move. Maintaining an authoritative state for other objects in the world is the task of the state distribution mechanism.

A straight-forward approach to ensure all nodes receive all relevant actions of players is broadcasting all actions. Broadcasting was used in MiMaze [49] and also in [89] but it is inherently not scalable. SimMud [80] adapted this approach to use multicast groups to broadcast messages only within a fixed size area. This approach only provides limited scalability as the number of players per zone may not be too big.

To exchange events more efficiently, later approaches focused on disseminating events only to players close enough to perceive the event. Only the players whose AOIs contains the player generating the event will receive it. Since AOIs typically have identical sizes, the relationship of being contained in another player's AOI is symmetric. Therefore, every player has to forward its events to all players within its AOI. To exchange messages efficiently, the overlay structure should reflect the locality in the virtual world by connecting nodes close to each other so messages can be transferred within a few routing steps. While there are approaches to use only weakly-structured overlays and gossiping for event dissemination [108], most approaches rely on specific overlay structures for event dissemination.

In Solipsis [78], nodes are connected to all other nodes within their AOI. Thus, events can be forwarded directly to other nodes. Since players are moving constantly, connections have to be adapted accordingly. The network has to stay globally connected and a node has to notice when a new node enters its AOI and connects to it. To ensure that also in cases when all nodes in the AOI lie in one direction or the AOI is empty, a node a maintains a second set of nodes outside its AOI. This set contains the closest nodes whose convex hull contains a . This ensures a node is always connected to outside nodes in every direction that can tell the node when a player enters its AOI. If connected nodes fail causing the node to lie outside the convex hull, it can issue a special query running in a circle around its position to discover nodes to repair the convex hull set. Nodes periodically exchange position updates to adapt the structure of the overlay.

In [77], a similar approach is used connecting to all nodes within the AOI and using an unstructured second set to discover nodes entering the AOI but without guaranteeing all nodes in the AOI will always be detected. pSense [105] works similar but global connectivity is insured by splitting the area around a node into eight sectors and connecting to the closest node outside the AOI in each sector. These nodes function as sensor nodes notifying nodes about players entering their AOI. Approaches like COVER [94] use an additional global quad tree of super-nodes responsible for maintaining player positions and detecting nodes entering AOIs. In QuON [8], locally constructed quad trees are used at each node to maintain the second set of nodes guaranteeing global connectivity.

Multiple approaches like VAST [69], VoN [68], Nomad [100], and others [19, 50] use Voronoi tessellations to create a Delaunay graph overlay structure by connecting neighbouring Voronoi cells. In these approaches, nodes are again connected to all nodes within their AOI and disseminate events directly. They connect to the nodes outside their AOI whose Voronoi cell overlaps their AOI. This structure automatically ensures a node is connected to other nodes in every direction to maintain global connectivity and detect nodes entering the AOI. These approaches mainly differ in how they adapt their structure to player movement, additionally predicting movement or using a red-black Delaunay structure that can be adapted faster [50]. In [74], an approach is presented to use the Delaunay graph to build a dissemination tree using other nodes in the AOI to forward an event to all nodes if the number of nodes is too high for the generating node to forward an event directly to all other nodes.

2.5 Cheating in MMVEs

Whenever written or unwritten rules govern human actions, there will be people trying to break these rules to get an advantage – reaching from just jumping the queue to committing crimes. Cheating in games is probably as old as gaming, especially when money can be won or lost, for example in poker. Even when nothing serious is really at stake, there will be people trying to cheat just to win. Adopting the definition of [127], we define cheating as:

Any behaviour that a player uses to gain an advantage over his peer players or achieve a target in an online game is cheating if, according to the game rules or at the discretion of the game operator (i.e. the game

service provider, who is not necessarily the developer of the game), the advantage or the target is one that he is not supposed to have achieved.

This definition has an important implication: whether some behaviour is considered to be cheating can be very different from game to game. It is up to the operator to define it as cheating or as valid behaviour. Nevertheless, there is a set of very similar rules that has emerged in MMORPGs. Thus, there are also some common types of cheating that have been analysed in [124, 126, 127]. We will present these types in the following section.

Considering the importance of competition and advancement as a motivating factor for playing MMORPGs [128], players will only play the game if it is fair: the rules are the same for everyone and nobody can break the rules. Having cheaters in the game will have severe consequences for the success of a virtual world [54]. Therefore, operators have to make sure cheating is not possible. We will show cheating countermeasures that have been proposed after discussing cheating types in the following section.

2.5.1 Cheating Types

There are two general types of rules a cheater can try to break. First, there are codified rules implemented as part of the game code defining what actions are possible and how the game world evolves according to these actions. They are usually not specified externally but the executable code running the virtual world defines these rules. Thus, breaking codified rules generally involves modifying either the code itself or changing the intended effects of the code by modifying its execution infrastructure. The cheats to break codified rules are specific for each virtual world and the way it is implemented. However, since current client/server-implementations work quite similarly, certain types of cheats have emerged.

In addition to codified rules, there are uncoded rules used by the game operator to specify how the game is supposed to be played. The game software itself is left intact but used in a way forbidden by the operator.

Collusion

A very common uncoded rule found in all competitive games is the prohibition of collusion. If multiple players are supposed to compete against each other but some of them collude, they might gain an advantage against non-colluding players. For

example, in World of Warcraft there is a ranking system for team-based player vs. player fights. Since each player can have multiple player characters, each team of players might form two teams: the first team consisting of characters supposed to be pushed up to higher ranks and the second team only serves as the victim for the other player's first team. This allows the winning teams to climb up the ladder very fast as matches also end very fast. This so-called win-trading is explicitly forbidden by the operator of World of Warcraft.

Automation

Stating the game has to be played manually by the player himself is another typical uncodified rule. Computers are in several ways superior to humans. Modern chess software beats most humans even on middle-class personal computers. In games like first-person shooters involving fast reactions and precision of player actions, players can use an aimbot – an additional piece of software generating mouse input for the game doing the aiming and shooting. In MMORPGs, there is also bot software completely automating the process of fighting NPCs by generating the necessary input for the game client. Thus, players can have the bot control his player character collecting in-game money and experience points to raise the level of the player character. It might sound strange for a player to use a piece of software to actually play the game rather than playing the game himself. However, some players consider the levelling or the farming of in-game money or crafting reagents as boring and try to automate this, giving them an advancement advantage over players playing manually.

Real-Money Trading

Using bots often appears in conjunction with another uncodified rule: the prohibition of real-money trading for in-game goods. There are companies offering players to sell them in-game money, items, or even high-level characters for real money. These companies either employ people mostly in low-salary countries to play the game and collect the items to be sold. However, some of them also use bots to automate the collection process. The time and the investment required to farm a certain amount of gold and the demand of players buying gold have led to the creation of real market prices for in-game money. This price allows determining the real value created in virtual economies – which can be substantial. In [23] the players

of the MMORPG Everquest [119] were shown to create a greater gross domestic product per capita than the people in Russia as of 1999.

Typically, the operator is not interested in letting other companies earn money with his game. Furthermore, the farming of in-game money leads to price inflation destroying the in-game economy as players not using bots or not buying in-game money from money-selling companies are not able to afford the prices of items, reducing their fun in playing the game. By allowing real money trading an operator would also indirectly admit that there is real value contained in virtual objects, establishing a notion of virtual property. This would have a lot of legal implications [10]. For example, if players were owners of a virtual item, the freedom of the operator to change or extend the game would be limited. Any adjustments to the power of the items or adding more powerful weapons to the game world would reduce the value of any existing now-weaker weapon. Owners of these weapons might demand compensation for the damage caused. Therefore, any trades involving real money are typically forbidden.

Abusing the Game Procedure

It is also forbidden to abuse the game procedure, for example when players about to lose a duel log out immediately to prevent their defeat. However, current games usually prevent this type of abuse because upon triggering a logout or disconnecting the player character usually stays in the world for a certain time. In effect, any kind of rule is possible: if the operator declares the game to be played with the left hand only, this would also be a valid uncoded rule – albeit a rule that does not make much sense and would be hard to enforce anyway.

State Tampering

The first way to break codified rules is to directly modify the state of the world, allowing to increase levels, power, and inventory of player characters or to weaken enemies. In [63] it was shown how a weakness of the client/server implementation in World of Warcraft could be exploited to modify the position of one's player character. In World of Warcraft, the complete player character state is stored at the central server and directly modifying this state is not possible because the server updates the state according to player actions only. It can check these actions and calculate their effects. There is one exception: the server trusts the client regarding the position of the player character. Its position is included in every action sent

to the server and the server updates the player character position to the reported value. Using a debugger, the memory at the address of the current position can be modified to reduce travel times or to always stay directly behind an opponent even when he is turning so he has no way to defend himself. While World of Warcraft uses techniques to make it harder to identify critical memory locations containing trusted state, e.g. dynamically changing memory locations, it does not perform any checks on the validity of reported position. Fundamentally, there is only one way to prevent the tampering with the world state in a client/server architecture: "Never trust the client!". The server should be the only one updating the world state and only according to actions of players that have been checked by the server.

State Exposure

Another attack on codified rules is the information or state exposure cheat [85]. Typically, a virtual world only allows a player to gain a limited amount of information about the world determined by the visual perception range. Having more knowledge might give a cheating player an unfair advantage over honest players. An example for this type of cheat is the so-called wallhack in first-person shooters. By changing settings in the graphics driver, the rendering of textures can be disabled for the walls in the game allowing cheating players to look through walls and see honest enemy players trying to hide. For this cheat, only the infrastructure of the game had to be modified, not the game itself. Similarly, so called maphack allow a player to read out the entire dynamically created map so he knows the whereabouts of all interesting objects. This is even possible without actually reading into the process memory by using network sniffers and constructing the map from the sniffed data instead. The underlying problem is again misplaced trust in the client. Once state is exposed to the client, the client cannot be trusted to keep it a secret.

Timing Cheats

Timing cheats are possible when players can get an unfair advantage by sending messages not at the time they are supposed to. The most important example is players getting an advantage by delaying sending their actions. This is only possible if the architecture actually allows such an advantage. In a simple client/server implementation, the server decides on the execution times of player actions based on the reception time of the action message and delaying a message would actually put a player at a disadvantage.

However, some implementations try to compensate the disadvantage of players with a high-latency Internet connection that always see the current state later than other players and have their actions always executed with a considerable delay. The architecture might synchronize clocks and put timestamps into the action messages of clients. Upon reception of such a message with an old timestamp, the server might conclude a player action happened earlier and recalculate the current state. Using timestamps is also mandatory in any peer-to-peer-based MMVEs as states will only be consistent if actions are executed at the same time which is impossible with variable message latency (see section 3.1.2).

Using timestamps opens up the possibility to delay an action or to replace the true time with an earlier timestamp. This lookahead cheat allows a cheating player to see the actions of other players and answer with an action put back in time so it will be executed first. To other people, this would appear as if the player actually performed this action first but it arrived later due to high message latency.

Inconsistency Cheats

In peer-to-peer-based MMVEs relying on exchanging player actions or state updates to calculate world state in a decentralized fashion, a cheating player can gain an advantage by tampering with this exchange. This is generally not possible in client/server architectures as nodes do not interact with each other but all messages go to and are processed by the central server.

A cheater could try to give invalid commands [124] by executing actions that are not allowed at a certain time because its precondition are not met. By disabling the local check of preconditions, a cheating player can send out the action anyway, hoping other nodes will execute it.

By creating inconsistencies in the world view of other players, a cheater might cause them to take wrong actions. He can send different actions to different other nodes causing their state to diverge. He can also completely blind specific players by suppressing the sending of any updates or actions to him.

A cheater might also cause other players to take certain actions by spoofing their identity directly creating actions in their name. If that is not possible, a cheater can still replay old recorded actions of other players.

Exploiting Bugs

A typical uncodified rule is the prohibition of exploiting bugs to get an advantage. Sometimes there are loopholes in the code, allowing players to use an uncommon combination of actions to create effects not intended by the developers. Examples for loopholes are sending negative amounts of in-game money to other players using the in-game money transfer, exploiting AI-errors causing NPCs a player could not defeat normally to become defenceless, or somehow being able to fly in areas where it is not possible.

However, a bug is also a codified rule. Since there is not external specification saying negative amounts of money may not be transferred or flying is supposed to be impossible, a player who exploits the bugs does not break a codified rule. Operators typically argue that players should have an idea about what is intended and what is not based on similarities to the real world or to other parts of the virtual world. Since transferring negative amounts of money is impossible in the real world, it should not be possible in the virtual world and players know that. Sometimes the distinction between bug exploitation and clever use of game mechanics is not that clear. For example in World of Warcraft, powerful guards prevent fighting in cities by attacking anyone who is attacking a player. A clever player could start a fight against an NPC outside the city and run into the city for guards to help him defeat the NPC. It is not immediately clear whether this is an exploit or a clever use of game mechanics. In the end, it is the operator's decision and his obligation to develop a consistent ruling on these border cases so players know what to expect.

2.5.2 Cheating Countermeasures

There are two fundamental ways to counter cheating: *prevention* and *detection*. Cheating prevention means cheating is essentially impossible. Either the system is just not vulnerable to a certain kind of cheat – like timing and inconsistency cheats that do not work in client/server-based MMVEs not using timestamps – or additional measures have been implemented making an architecture invulnerable against certain cheats that would have been possible otherwise. Countering cheaters by detection includes having a mechanism that can detect cheaters and, upon detection, punish them for cheating, for example by temporary or permanent exclusion from the game. If the detection works reliably, the number of cheaters will be reduced over time. Making the punishment visible to other players discourages cheating when they see there is actually a mechanism in place to catch cheaters [17].

Detecting and punishing cheaters can be used to counter all kinds of cheats. Breaking uncoded rules cannot be prevented but only detected and punished. The operator simply cannot prevent players from transferring real money outside the game exchanging it for in-game items or placing a robot in front of the computer to play the game. Breaking codified rules like state exposure can be prevented by limiting the amount of state given to clients [85] or can be detected, e.g. by tracing the view directions of players diverging for players using wallhacks as they tend to look at enemy players more often although these should not be visible for them [83].

A detection procedure faces the typical challenges of classification: working reliably, i.e. not generating false negatives while at the same time not classifying honest players as cheaters, i.e. not generating false positives. Internally, there is some kind of threshold to distinguish cheating from non-cheating behaviour. If detecting cheaters was a clear-cut case and performing a certain action under specific circumstances was cheating, that action could be denied in the first place preventing the cheating. Since falsely accusing honest players of cheating and punishing them will probably cause more harm to the success of the game than some cheaters going undetected, the operator will pick a more conservative threshold preferring false negatives over false positives. Cheaters will probably reverse-engineer these thresholds so a limited amount of cheating will always be possible with any detection procedure. Therefore, if prevention is possible it should always be used instead of detection.

The distributed storing and updating of peer-to-peer-based MMVEs introduces additional ways of breaking codified game rules like state tampering, state exposure, timing cheats, and inconsistency cheats compared to client/server-based MMVEs. In the following, we will show existing proposals to prevent these types of cheating.

Event Exchange

In the beginning, most proposals focused on preventing players from delaying messages or tampering with timestamps to see other player's moves before revealing their own moves. The lockstep algorithm [11] divided the game time into rounds. In each round, every player can perform one action he has to send to every other player. The round ends when the last player has sent his action. To prevent players from making their move for the round based on the other players' moves in the same round, each player first commits his move by sending out a secure hash of his move and reveals his move afterwards by sending out the original move only after receiving commitments from all other players. However, the highest latency between any pair of players dictates the round length determining the delay between a player

action and its execution. This will be too slow for fast-paced real-time games. Furthermore, any cheater can stop the progression of the protocol by not revealing his move.

Therefore, [31] proposed a pipelined lockstep protocol, allowing to interleave the committing and revealing of multiple actions. The rate at which actions can be generated can be increased. Still, the latency between triggering an action and executing it stays the same. This latency was reduced in NEO [48] explicitly bounding the length of a round. In NEO, each node must be able to send his move before the round end to at least half of the other nodes. After the round ends, all nodes send out their votes telling all other nodes which actions they received in time. Afterwards, all nodes will agree on which actions to perform. NEO still had some flaws allowing spoofing and replaying of actions as well as sending out conflicting actions to create state inconsistencies. SEA [30] extended NEO implementing the necessary cryptographic means to prevent these kinds of cheats. A different approach to prevent timing cheats is a hybrid architecture [27] using a central server. It first gives out encryption keys to encrypt all actions while decryption keys are given out synchronized to game rounds so no cheater can see a move earlier than the others and react to it. A decentralized approach based on the same idea was presented in [112]. In [25], one-time signatures were used for the same purpose.

There are also proposals to detect players modifying the timing behaviour of their messages by using trusted nodes to purposefully delay messages to nodes under test, trying to find out whether they are waiting for other players' moves [41, 42, 43]. However, these approaches failed to acknowledge that cheaters will react differently when noticing they are being tested. The underlying problem is that cheaters can always adapt their behaviour to any detection procedure in place.

Ensuring players send out their actions with the correct timing and do not send out conflicting actions is only one step to realize a fair game. It is still necessary to check whether the created actions are actually valid and to ensure the game state is only updated according to these actions and not tampered with.

Secure State Maintenance

The most common approach to check actions and ensure the game state evolves according to actions only is assuming the existence of trusted nodes known to work correctly. For example, RACS [123] uses a central server acting as a referee. It receives all actions of all players like the central server of client/server-based MMVEs

but does not send out updates to all nodes. They instead exchange actions and updates in a decentralized fashion and use the referee to resolve any conflicts that might occur. The referee is still the authority for the game state and stores it reliably.

Similar approaches distribute the game state among region controllers acting as a trusted authority for the state in a certain area [58, 75]. A hybrid architecture using untrusted region controllers is proposed in [72]. A central server checks whether the region controllers are working correctly, similar to RACS.

In [76], mutual checking was identified as an approach not relying on trusted nodes but only using the assumption that only a fraction of all nodes is malicious and most players do not cheat. Based on this assumption, [53] used a scheme of peer auditing to check the executed actions and the state distributed among untrusted nodes for correctness. However, it still relied on trusted nodes to resolve any conflicts appearing when audits failed. Therefore, the authority to decide and store the correct state are still trusted nodes.

An approach to store the world state in a distributed fashion while not assuming trusted nodes is Phitos [52]. It only used redundancy and replication to make sure the world state is not tampered with. However, it only ensures that state that has been stored is not modified afterwards. It does not show how to actually decide whether a state update was correct in the first place. As such, it can only be used as a kind of persistence layer as assumed by approaches like [75] using strategies to create state snapshots as proposed in [130].

Trusted Computing

A fundamentally different way to prevent cheating analysed in [76] is trying to make sure clients can actually be trusted and will not cheat. Approaches like Punkbuster [40] used in client/server-based online games monitor the game at the client trying to detect modification to the game software or additionally running software used for cheating. However, the underlying fundamental problem is the physical control of a user over his client computer. He can modify any software or modify its execution environment like system calls or other called libraries to change its behaviour in any desired way. All the server can do to check whether the client has been tampered with is send it a message and check whether the response message is the expected one. Given enough time, a cheater will be able to modify the program so it answers with the correct response.

However, using techniques of trusted computing like special hardware chips at the clients to create a trusted execution environment will make successful modification a much more costly endeavour [76]. Similarly, [26] proposed to use the isolated trusted execution processor of Intel's Active Management Technology platform to monitor the execution of the game software and detect cheaters.

In [93] using mobile guards with continuously changed protection algorithms is proposed to detect modifications. A mobile guard uses self-modifying code creating randomized checksums of the game code on the client while not leaving enough time for a cheater to analyse the behaviour of the guard and the currently calculated checksum.

Chapter 3

A Cheat-Resistant MMVE Architecture

So far, approaches to realize peer-to-peer based MMVEs have either not considered cheating at all or they relied on the existence of nodes trusted to be honest to deal with cheaters. We are interested in finding out whether it is possible to realize a peer-to-peer based MMVE based only on the assumption that a certain share of node tries to cheat while the majority of nodes is honest and follows the given algorithms. Our system should simulate one large virtual world with seamless movement instead of a world consisting of multiple disjoint areas.

Our goal is to obtain the same level of cheat-resistance that current client/server-based MMVEs offer. We do not consider breaking uncoded rules like automation or exploiting bugs. Our primary concern is to prevent cheaters from being able to tamper with the world state and to make sure the world state evolves only according to the rules of the world. Tampering is not possible in a client/server-based MMVE where the central server is the only authority for storing and updating the world state while the distributed storing in peer-to-peer-based MMVEs opens up that possibility. Furthermore, causing other nodes to perceive an incorrect world state by causing inconsistencies should be prevented like in a client/server-based MMVE where the central server is the only source for clients to perceive the world state. Modifying the timing behaviour or the timestamps of messages should also not yield an advantage for a malicious player. Finally, exposure of information about the world state should only be possible according to the limited perception range of players.

3.1 Basic Idea

State tampering is an inherent problem in all approaches based on loose consistency. All of them in some way update the world state on one node based on the world state reported by other nodes to resynchronize the world views. A node updating its state has to trust the reported state to be correctly calculated according to the rules only or it will update its own state based on tampered data.

The basic idea to solve this problem is to never update the state of a node based on the states of other nodes but have each node update its state only according to its own old state and the actions of players. If all nodes have the same state and receive the same set of player actions, they will all take the same decisions on which actions are allowed based on the current state and they will all calculate the same updated state. However, since a virtual world is a real-time simulation with continuously running updates, all the updates have to run synchronized among all nodes. This level of time synchronization cannot be achieved in practice. Furthermore, all nodes have to receive and process the player actions at the same time. This is also impossible due to varying message latencies in the Internet. However, using an optimistic event-based simulation instead of the standard update loop solves both problems.

3.1.1 Event-Based Simulation

Normally, a real-time virtual world is implemented using an update loop where the operating system triggers the world state update several times per second without giving any guarantees about the exact times. When the update is started, the current real-world wall time is read and the virtual world state is updated to that time.

A discrete event or event-based simulation models a system to perform state changes only at discrete points in time whenever an event occurs. Internally, an event-based simulator maintains an event queue containing all future events to be processed. It performs a loop dequeuing the first event from the queue, updating the state of the system, and adding any events generated during the update to the event queue. Since the simulated system is a physical system using wall time, each event has an associated timestamp signalling the time it will be executed to change the state in the system. The simulator always maintains a current virtual time [73] which is always equal to the timestamp of the event executed last.

If multiple nodes have the same system state and the same events, they will perform the same simulation creating the same result state. Therefore, implementing a real-time virtual world simulation using an event-based simulation can be used to synchronize the state on different nodes without exchanging the state itself.

However, the impression of real-time interaction is normally created using the fast-running continuous update loop. Therefore, we have implemented a hybrid virtual world simulation using a continuous update loop to internally perform an event-based simulation as proposed in [46]. On every update triggered by the continuous update loop, the simulator processes all events that have occurred since the last update applying their change to the world state timed to their respective execution times. However, there are also continuous changes in the world state as objects move. Therefore, before executing the event, the simulator first updates the state of all objects to the execution time of the event before applying the state change by executing the event. Finally, a throw-away copy of the world is updated to the current wall time and rendered on screen so the user always sees the current world state even when no events have occurred. Although performing multiple state updates in one invocation of the world update functions makes it much more expensive depending on the number of events to be processed since the last update, the effort is still negligible compared to the rendering of the world.

Using this hybrid model, actions of players are processed as events. A player starting to move would be realized by creating an event with the current time as timestamp and adding it to the event queue. Upon executing the event, the simulator will first update all objects including their current positions to the event's execution time and then change the movement vector of the player character. Upon execution of the next event, the player character position will be updated according to the movement vector and the time passed since the last updating of the player character. Its movement will continue with its position being updated until another event of the same player changes the movement vector again.

All other state changes have to be implemented using events, too. The collision detection has to be triggered by an event to stop moving objects colliding with other objects. Depending on the maximum movement speed, the detection events have to be executed several times per second to detect collisions reliably. The AI controlling NPCs has to inspect the virtual world and take decisions as a result of executing an AI event. If a player executes an action that has a certain cast time, the beginning of the cast will be an event. When the event is executed, it creates another event signalling the completion of the cast. Upon executing the completion

event, its effect will be applied. It could also create more events each applying their own effect or creating even further events. This way an action can cause multiple effects at certain times during its execution.

In our hybrid approach, the continuous update loop performs a kind of sampling of the event-based simulation. No matter how often or at which times the continuous update function is called, the updated state at all event times will always be the same. By exchanging events only and receiving them in time, all nodes in a network would be able to calculate the same state of the world. Luckily, events like the AI event, collision detection events, or events created as a result of executing another event do not have to be exchanged between nodes. Collision detection and AI events can just be scheduled at fixed intervals while events created by other events do not have to be exchanged as long as the root cause event is exchanged. In general, it is sufficient to only exchange events signalling the execution of an action by a player to realize a consistent distributed simulation.

3.1.2 Optimistic Distributed Simulation

For simulations on multiple nodes to stay consistent, the nodes must have added all events to the event queue before advancing the virtual simulation time and processing later events. Since transferring events over a network takes time, the execution time of an event should always be slightly in the future to leave room for all nodes to receive the event before its execution time. If a player presses the key to move forward, its character will not start moving immediately but an event will be scheduled with a future timestamp. This local lag can typically be in the range of 150 ms without harming the perception of players to interact with a real-time world [89].

However, these 150 ms might not be enough to always transfer the events in time since some messages might experience higher latency and clocks are not perfectly synchronized as well. Under these circumstances, a node will receive a late event which means the state it calculated is incorrect as it is missing the late event. There are two fundamental ways to handle late events: conservative lock-step synchronization [91] and optimistic techniques like Time Warp [45, 73].

In a conservative algorithm, all nodes first form an agreement to advance the current virtual time by explicitly committing themselves to not generate an event before the time in question, effectively preventing late events. Since reaching this agreement takes a considerable amount of time, conservative algorithms are usually unsuitable

to implement a real-time simulation as time in the world could not advance while the agreement is in progress.

In Time Warp, every node advances its virtual time with the current real time without waiting for an agreement with other nodes. In case an event is late, it repairs its incorrect state. Every node creates a copy of its current state at checkpoint times. Furthermore, it does not remove processed events in the event queue. Upon receiving a late event, the node performs a rollback to the latest checkpoint before the event and re-executes all events including the late event after the checkpoint. This way Time Warp allows rewinding and replaying the world simulation at any point in time. The rollback and the reprocessing of events will all happen within the same update so the user will not notice the replaying. Instead, he will see a sudden change in world state as for example a standing player is suddenly teleported forwards when a node notices it started moving a second ago. Therefore, rollbacks may cause temporary inconsistencies with instant world state changes not covered by the rules. However, in combination with local lag late events should be the exception and the world state will be eventually consistent when all late messages have arrived.

3.1.3 Limitations

Combining Time Warp and local lag to realize a decentralized MMVE has been proposed in [89] and similarly in [32] using it to implement mirrored game servers. Exchanging only player actions resulting in a consistent simulation of the world state automatically ensures the world state only evolves according to the rules of the world as we have shown in [107]. Therefore, it is generally more suitable to prevent cheating in a peer-to-peer-based MMVEs as discussed in [29]. Since all nodes store the complete state of the world, a joining node will also be able to retrieve the correct state of the world. It asks all nodes for their state and performs a voting on the results. Since the current state of a node can always be rolled back, sparse vector clocks [125] can be used to identify which nodes have included which player actions to identify a consistent state. As long as the majority of nodes are honest, it will start the simulation based on the correct state.

However, there are two major limitations of using this approach. First, it is very prone to timing cheats with cheaters delaying their actions purposefully causing rollbacks at honest players' nodes. This causes honest players to take actions based on a state that will be corrected later on. It is also vulnerable to inconsistency cheats by malicious players sending different actions to different nodes or not sending actions

to some nodes. This will cause the world state to diverge. To fix this, all honest players would have to send out any actions they received from other players a second time to make sure all honest players receive all actions sent out by malicious nodes.

The second and much more severe limitation is the lack of scalability. All nodes perform replicated simulations of the whole world and all actions performed by players have to be sent out to all other nodes. If inconsistency cheats should be prevented each action actually creates n^2 messages. Using this fully-replicated simulation would only be possible if the virtual world consists of many disjoint areas with a tight limit on the number of players per area. A single large virtual world with a massive number of players cannot be realized this way. The key to scalability is to distribute the world state among multiple nodes and have each node simulate only a part of the world. However, this means taking over the world state from other nodes will be necessary in any scalable solution as nodes move around and simulate the state in different areas. This fact inspired our cheat-resistant architecture described in the next section.

3.2 Architecture

In a scalable architecture, nodes can only simulate a part of the virtual world. To present the world view to the user, they will have to simulate the area surrounding the player. If the player moves to a different area, its node will have to simulate the world in that area. Therefore, it has to take over state from nodes currently maintaining that area. If these nodes are malicious, they can report a tampered state. Even assuming an honest majority and using voting would not help, as a cheater could have multiple player characters under his control and move them to the area to get a local majority. A node needs a way to ensure state it has received from other nodes was correctly created according to the rules.

Therefore, we propose to use a hybrid architecture consisting of two disjoint parts. First, there is the decentralized simulation of the virtual world. It may be based on approaches like Solipsis [78], pSense [105], or VoN [68] using a peer-to-peer overlay adapting its network structure to the current player positions allowing to exchange events and state updates locally. These approaches do not consider cheating so cheaters might report tampered state. Therefore, there is a second overlay structure to realize a reliable distributed storage. This storage fulfils two tasks. As we have shown in [64], it acts as a reliable persistent storage to ensure world and player character state is not lost when nodes join and leave. It also maintains the current

state of the world ensuring it only evolves according to the game rules to protect it from tampering by malicious nodes.

The tools to achieve this reliability and resilience are replication of data, redundancy of simulation, and use of voting to establish the correct state with a high probability. While the first part of our architecture, the decentralized simulation of the virtual world, has to react fast to player input and continuously update the state, the storage can take its time to perform votings and focus on state correctness to prevent tampering. However, due to the voting the storage will only be able to provide a probabilistic protection against tampering and data loss. With increasing churn or increasing number of malicious nodes, the probability for data being lost or getting tampered will increase. This probability can be reduced by increasing the amount of redundancy in storing and updating the world state. On the other hand, increasing redundancy increases costs in terms of bandwidth consumption or computational power. Our goal is to find out what level of reliability and protection can be realized at what costs.

There are already approaches like PAST [104] and OceanStore [82] to realize a reliable distributed storage based on a peer-to-peer overlay, using replication to provide reliability and different techniques to keep replicas consistent. They are, however, unsuitable to be used as a reliable data store for an MMVE. First, they use hashing to distribute data on nodes destroying any locality of data. Objects close to each other in the virtual world will not end up on the same or close nodes but they will be distributed all over the network. To update the state in an area, all objects in that area must be requested. Retrieving them from all over the network will be much more expensive than retrieving them from a few close nodes. Second, these distributed storages just store data and make sure it is not lost. There are no means to make sure the stored data is actually correct.

Therefore, we have designed a reliable distributed storage to persistently store data in a virtual world. It is based on a structured peer-to-peer overlay mapping the locality in the virtual world to locality in the network to allow efficient retrieving of close data. Furthermore, the storage executes an update procedure calculating the correct state with a high probability despite node churn and the existence of malicious nodes.

3.3 System Model

Our system consists of Internet-connected PCs to become the nodes of our peer-to-peer overlay. The PCs have sufficient amounts of storage and computation power to store data and perform simulations. We expect the available bandwidth to be the limiting factor. We assume all nodes are able to connect to each other using a link layer implementation like the one described in [65]. It allows establishing connections even in the presence of NAT (Network Address Translation) routers normally not allowing incoming connections. The established connection is reliable and message order will be preserved as realized by TCP. However, message latency might vary and be high in case of message loss on the underlying layer.

Nodes form an overlay so every node is only connected to a subset of all nodes on the link layer. However, on the overlay layer all nodes can still exchange overlay message using multi-hop routing. Nodes are considered to be unreliable. They might fail unexpectedly at any time. Churn is high so nodes are joining and leaving the overlay constantly. The system is not an open system. There is an operator of the virtual world who also designs and implements the virtual world and lays out its rules. He also operates a trusted central certification authority as we proposed in [121, 122]. Players have to acquire a certificate from that certification authority to join the overlay. Using this certificate, integrity of messages and non-repudiation can be realized. Furthermore, players identified as cheaters can be punished by revoking their certificate. To make sure they do not just come back with a new identity, acquiring a certificate must be expensive, e.g. by costing money or by being bound to real-world identities.

At any time, only a subset of all players is in the network. The majority of these players is honest and their nodes follow the protocol specified and implemented by the operator. However, players have full physical control over their computer. Therefore, an attacker can tamper with his node to not follow the protocol in any way. He can modify the code, modify the state of memory, or send out any kind of message at any time. While the world is running, there are no nodes in the network that can be trusted to follow the correct protocol.

We use physical clocks to control the timing behaviour of the update algorithm and to put timestamps into messages signalling the execution times of player actions. We assume clocks among honest nodes to be synchronized down to ten milliseconds, a value than can be achieved over the Internet by using the Network Time Protocol (NTP) [90]. Furthermore, we assume drift of clocks and timers to be bounded.

Chapter 4

A Self-Stabilizing Delaunay Overlay

This chapter presents the overlay we developed as base for the reliable storage. First, we analyse the requirements of the overlay. Then we show how we designed the overlay to fulfil these requirements. We evaluate how the overlay performs with respect to our requirements in chapter 6.

4.1 Requirements

The purpose of the storage is to reliably store location-dependent data. In our case, this is data with coordinates from a 2D Euclidean space as key. To allow storing and retrieving of data, the overlay has to support communication between nodes based on locations. Two functions for message exchange have to be supported:

Unicast Communication The overlay must route unicast messages from one node to one destination location until it is received by the node responsible for that location. This type of message will be used by the storage to store data at other nodes. It will also be used to retrieve data for a given location and return it to the requesting node.

Multicast Communication Multicast messages must be routed from one node to a given area. All nodes responsible for data in that area will receive this message. This type of message will be used by the storage to retrieve all data in an area.

To realize our goal of a reliable, resilient, and scalable storage, the overlay has to fulfil the following non-functional requirements:

Scalability The overlay has to be scalable. Ideally, the traffic at each node should not depend on the number of nodes in the network. This means nodes can be added to the overlay without incurring significant additional load on other nodes and the system could run with an arbitrary number of nodes. To realize this, the overlay needs a structure allowing efficient routing of messages and a scalable mechanism to maintain that structure.

Reliability The overlay must work reliably despite the unreliability of individual nodes. If a node fails, the maintenance mechanism has to repair the network structure so the overlay can continue to exchange messages on behalf of the storage.

Resilience The overlay must be resilient to malicious nodes not following the protocol of the overlay e.g. by dropping or tampering with messages. Specifically, the overlay must prevent malicious nodes from gaining more influence than proportional to the share of malicious nodes in the network.

4.2 Overlay Structure

Nodes can use the overlay to send messages to other nodes using their 2D overlay addresses. The overlay will route these messages via intermediate nodes if necessary. However, the overlay also allows finding nodes responsible for keys from the given key space. It performs a distributed computation of a mapping from a key space to the set of nodes in the network.

4.2.1 Key Space and Node Responsibility

Overlays like Chord [113] and Pastry [103] use this ability to realize a distributed hash table (DHT). A DHT allows to store data under arbitrary given keys by hashing these keys to a numeric key space $K = \{0, \dots, k_{max}\}$ with $k_{max} = 2^l - 1$ and l being the length of keys in binary representation. The typical way of mapping keys to nodes is by assigning node IDs from the same key space K . Then, a distance function $d : K \times K \rightarrow \mathbb{R}_{\geq 0}$ is used to compute distances from node IDs to data keys. The node with the smallest distance to the data key is the responsible node for this key. Pastry uses the metric distance function $d_p(a, b) = |a - b|$. Therefore, a node b with a being the node with the next smaller ID (*predecessor*) and c being the node with the next bigger ID (*successor*) is responsible for the interval $[\frac{a+b}{2}, \frac{b+c}{2})$. With

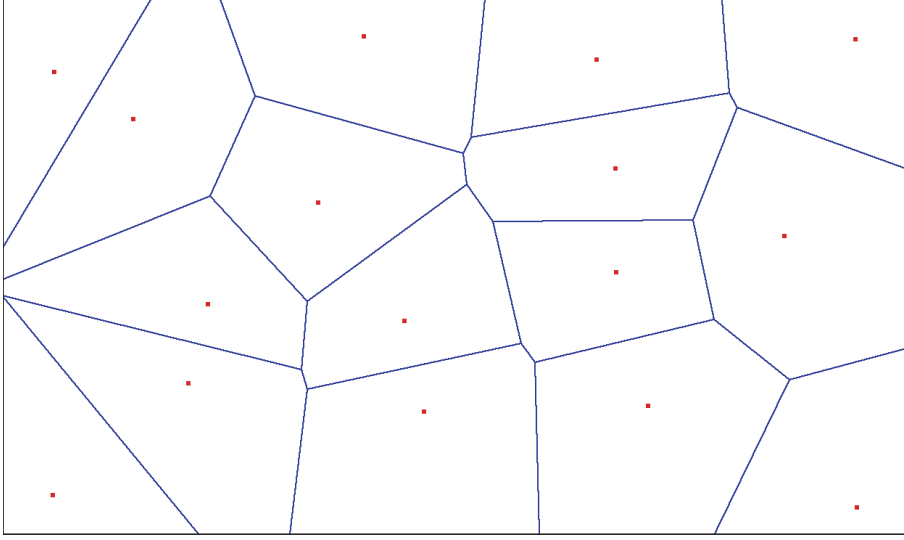


Figure 4.1: Voronoi tessellation

Chord's distance function $d_c(a, b) = \begin{cases} a - b, & \text{if } a > b \\ k_{max} - b + a, & \text{else} \end{cases}$ a node is responsible for all keys between itself and its predecessor if a is used as node key and b is used as data key.

The fact that nodes are responsible for their closest keys allows a greedy routing procedure if the network is structured appropriately. When a node receives a message with a key as destination, it is either the closest node and receiver of the message or it knows another node that is closer to the destination and can forward the message to that node.

In contrast to DHTs, our system uses two-dimensional keys $k = (k_x, k_y) \in K \subset \mathbb{R} \times \mathbb{R}$ to represent the location of objects in the virtual world as points in a plane. Following the DHT approach, we assign node IDs from the same space. We use the Euclidean metric $d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$ as distance function. If $N \subset K$ is the set of node IDs, the Euclidean metric creates a *Voronoi tessellation* [7] of the plane K , see figure 4.1. For each $n \in N$ its *Voronoi cell* contains all points k where k is closer to n than any other element of N . Two adjacent Voronoi cells with a common border are called *Voronoi neighbours*. For points on the border with equal distance to multiple Voronoi neighbours, we defined the leftmost neighbour and for nodes with equal x -coordinate the topmost neighbour to be responsible for these points. The tessellation of the key space of Pastry into disjoint intervals is actually the one-dimensional case of a Voronoi tessellation.

Mapping keys to closest nodes also allows efficient area query processing. If a node is responsible for one key of a given requested area, it is probably also responsible for other close keys of this area and so these keys can be returned in one message.

4.2.2 Node ID Assignment

The two-dimensional ID of a node determines which data the node is responsible for. Therefore, a node must not be allowed to choose its own ID. Otherwise a cheater could easily place nodes under his control in locations allowing him to tamper with stored data. Instead, node IDs are assigned by a certification authority (CA). Nodes exchange their certificates upon connection establishment and verify their node IDs using the public key of the CA. Furthermore, a price has to be associated with obtaining an ID as shown in [44] using game theory to prove the negative effects of free identities. Otherwise an attacker could just obtain a large number of identities to launch a Sybil attack [36] and dominate the overlay. The price makes obtaining IDs expensive and limits the number of IDs a cheater can have. This is also necessary to punish cheaters by excluding them through certificate revocation. Otherwise he could just return with a free new ID once his malicious behaviour was detected. This price might be a usage fee. It could also be binding a certificate to a credit card number or a verified real person identifier.

4.2.3 Primary Structure

The primary structure of a network is a graph with nodes as vertices of the graph and edges representing connections between nodes. If a node a and b are connected then a is a neighbour of b and vice versa. The network should be structured in a way allowing a simple greedy routing procedure: a message for a destination is forwarded from one node to a node closer to the destination until the closest node is reached. A node only needs local information – its neighbours – to decide who to forward the message to. This only works when the structure does not contain different nodes with local distance minima to a destination. If a node calculates its distance to a destination and compares it with the distances of its neighbours and concludes it is the closest node there may not be a different node drawing the same conclusion for this destination. Otherwise messages for a destination will end up on different nodes, depending on where they came from.

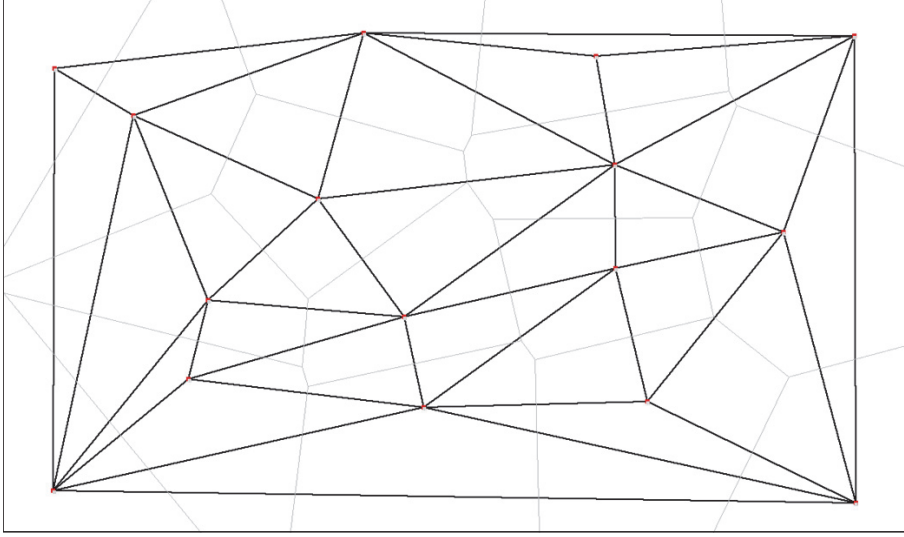


Figure 4.2: Delaunay graph

Chord and Pastry both use a simple ring structure without different local distance minima. In Chord, a node's successor is its only neighbour. In Pastry, successor and predecessor are neighbours of a node. Since their key space K is a ring of congruence classes \mathbb{Z}/k_{max} both form a ring structure closing the ring by connecting the node with the highest ID with the node with the lowest ID. In Chord, the ring is a directed graph. When performing the greedy routing, every node can only decide if it or its successor is closer to a destination. For a ring of n nodes, a routed message is passed around the ring. Since nodes are essentially ordered, it is guaranteed that a message arrives at the responsible node. Pastry's ring is undirected and messages can be routed in both directions around the ring. Again the ordering of nodes guarantees the responsible node is reached eventually.

In our overlay, we chose the Voronoi neighbours of a node to become its overlay neighbours. Connecting the Voronoi cell centres to their Voronoi neighbours creates a *Delaunay graph* [7], see figure 4.2. A Delaunay graph is a triangulation of the Euclidean plane and the dual graph to a Voronoi tessellation.

Unicast Message Routing

When a node routes a message, it will compare its distance to the message destination to the distances of its neighbours. If it is closest, the destination lies within its Voronoi cell and it is responsible. If not, the node will forward the message to the closest neighbour which will perform the same greedy routing procedure. As the Voronoi tessellation covers the whole plane and all Voronoi neighbours are

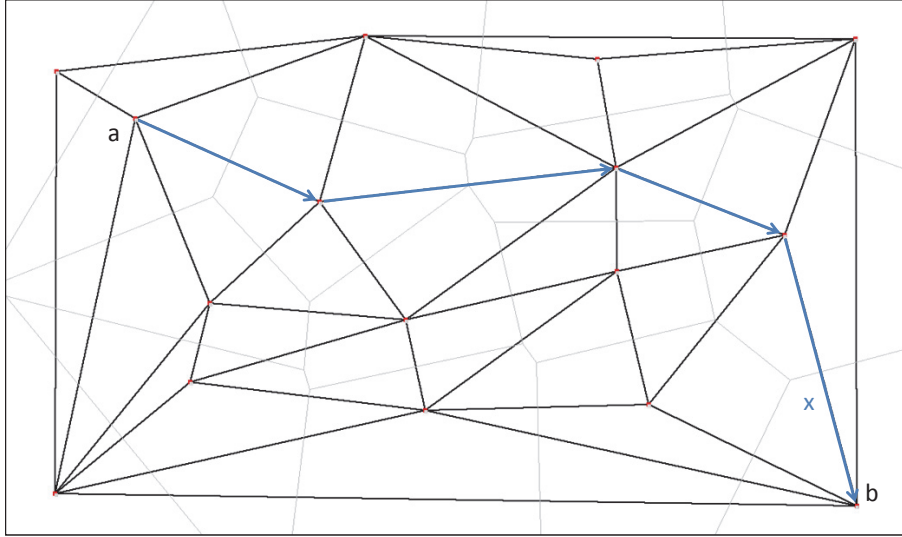


Figure 4.3: Neighbourhood routing in the Delaunay structure

connected, each routing step brings the message closer to its destination until the responsible Voronoi cell is reached. Figure 4.3 shows the route of a message sent from a node a to destination x lying in the Voronoi cell of b . This is a worst-case scenario as source and destination have the maximum possible distance along the diagonal of the key space. With even distribution of nodes in the key space, the worst-case routing performance in the Delaunay structure is $O(\sqrt{n})$ due to the two dimensions of the key space. Compared to the $O(n)$ performance of Pastry with its one-dimensional ring, the Delaunay structure allows getting closer to a destination in two dimensions simultaneously on each hop.

4.2.4 Routing Table

In a one-dimensional overlay structure like a ring, routing needs $O(n)$ hops on average. This linear routing performance does not scale in general. Assuming every node sends one message in a given time frame, each of which will be forwarded $O(n)$ times by other nodes then $O(n^2)$ messages will be transferred on the link layer. On average, every node will have to transfer $O(n)$ messages. This means the traffic on one node increases linearly with the number of nodes.

Therefore, peer-to-peer overlays usually feature an additional structure connecting nodes far away from each other in the key space. This allows a node to bridge a much larger key space distance in one hop when routing a message. Thus, routing

Node ID = 013202				
	C ₀	C ₁	C ₂	C ₃
R ₀		102310	213021	330122
R ₁	003123		022300	031003
R ₂	010323	011231	012331	
R ₃	013011	013112		013321
R ₄		013210	013223	013230
R ₅	013200		013202	

Figure 4.4: Pastry routing table, $b=4$, $l=6$

paths become shorter and the traffic per node is reduced. For example, Chord and Pastry both feature a routing path length of $O(\log(n))$.

Each node stores the set of nodes providing these additional long distance connections in a routing table. Our routing table is heavily inspired by the routing table of Pastry adapting it for two dimensions. Pastry features a two-dimensional routing table with b columns and l rows where b is the radix used in the key space and l is the length of keys in digits. The size of the key space is $N = b^l$. A routing table entry $e_{i,j}$ is defined by its column i , $0 \leq i < b$ and its row j , $0 \leq j < l$. For a node c with ID $c_0c_1\dots c_{l-1}$ and a node d showing up in entry $e_{i,j}$, d 's ID $d_0\dots d_{l-1}$ has to fulfil a certain constraint. It must match c 's ID in the first j digits $c_0\dots c_{j-1}$. Then, digit j must equal i . The remaining digits are not constrained. Therefore, d 's ID must be of the form $d = c_0\dots c_{j-1}i*$. In the first row with $j = 0$, the constraint effectively means that a node in column i starts with i . All nodes in the second row start with c_0 and have the column number i as the second digit. All nodes in the third row start with c_0c_1 with the column number i as the third digit. The entry $e_{c_j,j}$ is left blank for all $j = 0\dots l-1$. So in every row j the column number equalling the node ID in digit j is not occupied. This entry would contain a node with ID $c_0\dots c_{j-1}i$. Since $i = c_j$, it would be $c_0\dots c_{j-1}c_j$. However, nodes matching c 's ID in the first $j+1$ digits are found in the next lower row $j+1$. Therefore, the entries $e_{c_j,j}$ are not needed. Entries can also be empty if no node with a matching constraint exists. This is highly likely for lower rows as the key space is usually only sparsely populated with nodes. Figure 4.4 shows an example of a Pastry routing table.

This kind of routing table allows a prefix routing scheme to be used yielding $O(\log(n))$ routing performance. For a message destination $t = t_0\dots t_{l-1}$, even in the worst case c will have a routing table entry matching t in the first digit because c has nodes in its first row with all possible digits. Assuming d is that entry, d 's ID will be $t_0d_1\dots d_{l-1}$. Now d will also have an entry in its second row matching t in the first two digits because d has all possible digits in its second row. This means even in the worst case, this routing procedure allows a message to get closer to its destination with

Node ID = (013202;301230), Layer = 0				
	C ₀	C ₁	C ₂	C ₃
R ₀	(013120;021321)	(032201;120103)	(001302;232201)	
R ₁	(100332;032201)	(102311;133021)	(132210;221023)	(102303;312032)
R ₂	(203120;021123)	(212031;120322)	(202310;213021)	(210310;330210)
R ₃	(322123;032112)	(332012;123210)	(302101;210320)	(310120;320120)
Node ID = (013202;301230), Layer = 1				
	C ₀	C ₁	C ₂	C ₃
R ₀	(001312;302132)	(003201;312003)	(000302;322201)	(000032;332201)
R ₁		(010231;313302)	(012210;322023)	(012303;331032)
R ₂	(023120;301123)	(021201;312322)	(020230;321321)	(021010;333010)
R ₃	(032213;302112)	(033012;312210)	(030101;321032)	(031010;332012)
Node ID = (013202;301230), Layer = 2				
	C ₀	C ₁	C ₂	C ₃
R ₀	(010312;300212)	(010301;301203)	(010032;302201)	(010012;303210)
R ₁	(011002;300301)	(011021;301330)	(011210;302023)	(011303;303103)
R ₂	(012320;300123)	(012120;301322)	(012031;302132)	(012010;303010)
R ₃	(013223;300112)		(013210;302103)	(013100;303212)

Figure 4.5: Three layers of a routing table with $b_x = 4$ and $b_y = 4$

each hop matching the destination in one more digit. The maximum number of hops is l . With $l = \log_b N$, the worst case routing performance depends logarithmically on the size of the key space. If the n nodes are evenly distributed in the key space, routing will take $O(\log_b n)$ steps.

Applying this scheme to two dimensions, we use a three-dimensional routing table with b_x columns, b_y rows, and l layers. Figure 4.5 shows how the first three layers contain nodes from different areas of the key space. For our key space $K = K_x \times K_y$, b_x is the base of the numbers used for the integer part of the x -coordinate and b_y is the base of the numbers used for the integer part of the y -coordinate. The length of the x - and y -coordinate integer parts is l . Therefore, an ID $d \in K$ has the form $d = (d_{0,0} \dots d_{0,l-1}.*; d_{1,0} \dots d_{1,l-1}.*)$ with $d_{0,0} \dots d_{0,l-1}.*$ being the x -coordinate and $d_{1,0} \dots d_{1,l-1}.*$ being the y -coordinate of d . We denote the fractional part of the IDs as $.*$. However, in the following, we will only consider the integer part of coordinates as the fractional part is only used when switching to the neighbourhood-based routing. Also, we assume keys already differ in the integer part so no keys have equal integer parts. This can easily be ensured by the central certification authority generating the keys. Consequently, the size of the key space using only the integer part is $|K| = |K_x| * |K_y|$ with $|K_x| = b_x^l$ and $|K_y| = b_y^l$.

The node ID d in an entry $e_{i,j,k}$ of node c with ID $c = (c_{0,0} \dots c_{0,l-1}; c_{1,0} \dots c_{1,l-1})$ must fulfil a constraint similar to Pastry. The x -coordinate $d_{0,0} \dots d_{0,l-1}$ must match c 's x -coordinate in the first k digits. The value of digit $d_{0,k}$ must equal i . The y -coordinate $d_{1,0} \dots d_{1,l-1}$ must also match c 's y -coordinate in the first k digits. The value of digit $d_{1,k}$ must equal j . Thus, d must be of the form $d = (c_{0,0} \dots c_{0,k-1}i*; c_{1,0} \dots c_{1,k-1}j*)$ for $k > 0$ and $d = (i*, j*)$ for $k = 0$.

Unicast Message Routing

This routing table enables prefix routing with logarithmic routing performance. Figure 4.6 shows a node a sending a message to destination x . In this example, both bases are identical with $b_x = 2$ and $b_y = 2$. The ID of node a starts with 0 in the x-coordinate and 0 in the y-coordinate. Node a lies in the *zone* $Z_{0,0}$. A zone $Z_{i,j}$ is a rectangular sub-area of the key space where all keys start with i in the x-coordinate and j in the y-coordinate. Zones can be divided into sub-zones. Zone $Z_{ik,jl}$ is a sub-zone of zone $Z_{i,j}$ with all keys in $Z_{ik,jl}$ starting with ik in the x-coordinate and jl in the y-coordinate.

The way the routing table is built, a has stored one node from each of the zones $Z_{0,1}$, $Z_{1,0}$, and $Z_{1,1}$ in the top layer of its routing table. The destination x is not in a 's zone $Z_{0,0}$ but in zone $Z_{1,1}$. Therefore, a will forward the message to the node b in $Z_{1,1}$ stored in its top routing table layer. Node b shares a common prefix with the message destination x . Both start with 1 in the x- and y-coordinate which means both are in zone $Z_{1,1}$. However, they are not in the same sub-zone. Node b 's ID starts with 10 in the x-coordinate and 11 in the y-coordinate. Thus, b is in sub-zone $Z_{10,11}$ while the destination x is in sub-zone $Z_{11,11}$. In the second layer of its routing table, b has stored nodes from each of the other sub-zones $Z_{10,10}$, $Z_{11,10}$, and $Z_{11,11}$. Since x is in $Z_{11,11}$, b will forward the message to c taken from the second layer of its routing table. Node c is situated in sub-zone $Z_{110,110}$ while x is in $Z_{111,110}$. Again, c has one node from each of the zones $Z_{110,111}$, $Z_{111,110}$, and $Z_{111,111}$ in the third layer of its routing table. Therefore, it will forward the message to node d in zone $Z_{111,110}$. Assuming the integer parts of IDs has three digits (the routing table has three layers), this routing process will finish here and d will continue with the neighbourhood-based routing in the Delaunay structure. It will compare its distance to x with the distances of its neighbours and find out it is the final receiver of the message.

In each routing step, the next node matches the message destination in at least one more digit in the x- and y-coordinate. Even in the above worst case, when sending node and destination share no common prefix, only l routing steps will be necessary. From l being the length of the integral part of keys and $N = b_x^l * b_y^l$ being the size of the integral part of the key space follows $l = \log_{b_x * b_y} N$. With even distribution of n nodes in the integral part of the key space, the worst-case routing performance is $O(\log_{b_x * b_y} n)$.

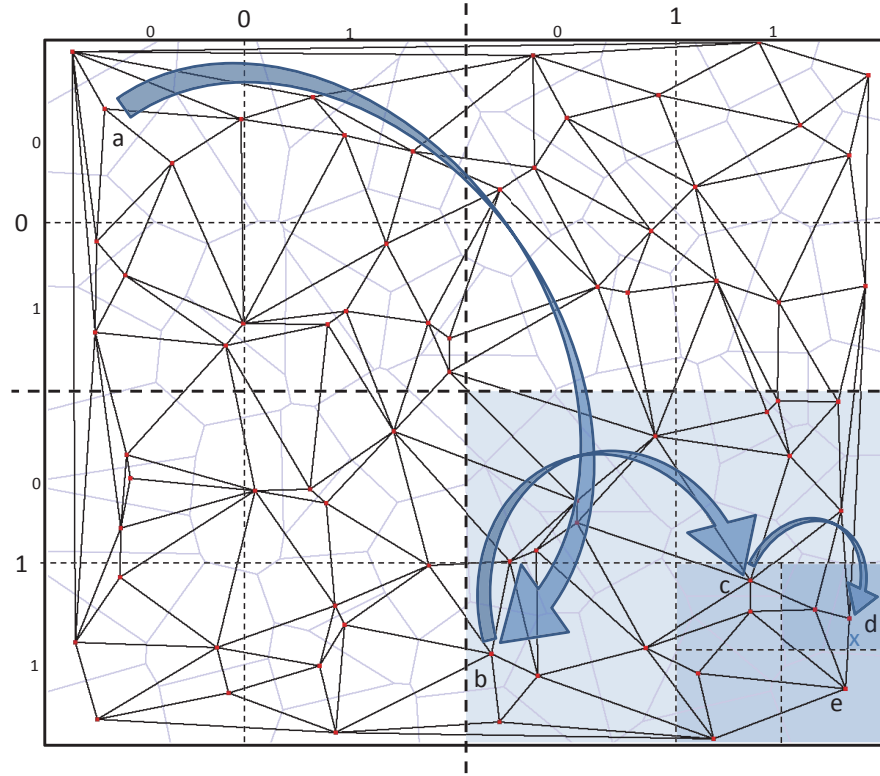


Figure 4.6: Message routing using routing tables

4.2.5 Shortcuts

In addition to the neighbourhood sets and the routing table, we also maintain a third set of nodes called shortcuts. It serves as a cache containing additional nodes a node often communicates with. This is especially useful for answering queries in the storage. When a node receives a request routed over intermediate nodes, it will send a reply back. By putting the original requester into the shortcuts set and establishing a direct connection to him, we can send the reply back directly and do not need to route it. The route back would probably be different from the first route, giving an attacker more chances to tamper with the query processing.

Furthermore, when more queries from the same source arrive, the already established shortcut connection can be used to answer it. To do this, shortcuts are also included in the routing procedure. Whenever a shortcut is closer to a message destination than any neighbour or routing table entry, the message is forwarded to the shortcut instead. Getting closer to a destination than normally possible is always an improvement. It creates shorter paths. Shorter paths mean smaller latency and fewer chances for a malicious node to tamper with the routing. Since a query reply

message is addressed directly at the requesting nodes position, the shortcut will always be responsible and the reply will be transferred back directly.

4.3 Delaunay Structure Maintenance

The maintenance algorithm must maintain the overlay structure as nodes join and leave the network. Furthermore, it must contain the influence attackers have in the network. To tamper or drop more messages, attackers will try to have more messages routed via malicious nodes. This means malicious nodes appear more often in the routing tables and neighbourhoods of other nodes than expected considering their share in the network. The maintenance algorithm must prevent this and keep the share of malicious nodes in routing tables and neighbourhoods proportional to their total share in the network.

4.3.1 Self-Stabilization

Typically, proposals for peer-to-peer overlays describe their maintenance algorithm using procedures for joining and leaving of nodes. Given an existing overlay, they show how one node joins the overlay e.g. by routing messages to its new neighbours and exchanging messages with them to be integrated into the structure as their new neighbour. Often the routing table is also filled while executing the join procedure. There might also be a mechanism to adapt the structure for graceful leave or when neighbours detect the ungraceful leave of a node.

When we started developing our overlay, we discovered two flaws in these approaches. First, the procedures given usually assume an existing overlay but they do not work when the overlay is started for the first time. In that case a lot of special conditions can arise that have to be treated differently. For example, a protocol specifying left and right neighbour in a ring have to be informed does not work as expected if left and right neighbour are identical because only one node is already there. Pastry uses a set of multiple neighbours in both directions on the ring for robustness reasons, making it hard to deal with these special cases arising when the overlay is not already sufficiently large.

Second, these procedures usually assume only one local change in the network at a time. Two join procedures can be executed concurrently when they happen in different locations. In the same neighbourhood two concurrent joins lead to the two

independent join procedures interfering and in result the overlay structure might become inconsistent. To solve this, Chord specifies the case of concurrent join of two neighbours. However, there might be even more concurrent joins or joins interfering with leaves. Although concurrent joins are unlikely to happen when the overlay is large, they are common in the early phases of starting up the overlay when it is still small.

Self-stabilization [35] is a principle guaranteeing a system to reach a legitimate state in a finite number of steps from any starting state using "distributed control". Every component only acts on state it sees locally and the system as a whole finally reaches the desired global state. Applying this principle to maintain an overlay structure, nodes only need to exchange local information about their state with their neighbourhood. A self-stabilizing overlay is automatically able to deal with nodes joining and leaving as it will reach the desired state from any state. A joining node just connects to any node of the network and it will automatically be integrated at the correct position after a finite amount of steps. A leaving node just closes all connections and the overlay structure will adapt. The algorithm only requires the network graph to be connected. Otherwise, it will create partitioned overlays each with its own structure.

Decentralized self-stabilizing algorithms have been used to create trees [47] or a variant of Chord using a self-stabilizing ring [79]. In [71] a self-stabilizing algorithm is proposed to create a Delaunay graph. It is a synchronous algorithm proceeding in rounds. In every round, nodes exchange information about their neighbourhoods and use this information to form new connections adapting their neighbourhood. This algorithm features a guaranteed worst-case number of rounds of $O(n^3)$ to create a Delaunay graph from any connected starting graph.

However, in practice this algorithm has two drawbacks. First, the round length would have to be picked conservatively to allow information exchange and connection establishments in an Internet setting. This might lead to long stabilization times in the worst case considering the $O(n^3)$ bound on the number of rounds. Second, the algorithm needs regular information exchange even if no changes occur in the network. This is not efficient when only few local changes happen in the network.

Therefore, we have developed an asynchronous self-stabilizing algorithm to create a Delaunay overlay. Its basic idea is to exchange neighbourhood information with other nodes whenever the neighbourhood changes. Nodes use a message called *NetworkView* containing the source of the message and its neighbours to exchange

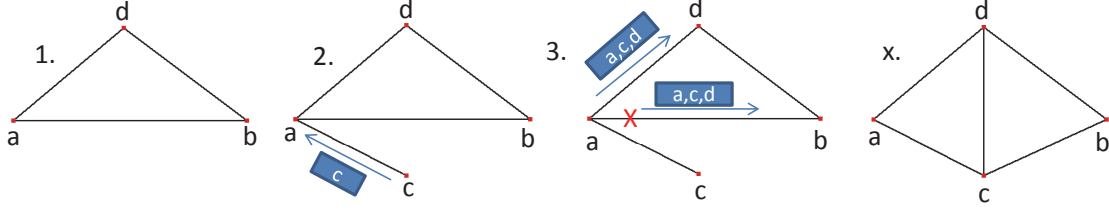


Figure 4.7: Sending a NetworkView before disconnecting an ex-neighbour

this information. Listing 4.1 shows a simplified version of the procedure executed when a NetworkView is received.

4.3.2 Adding and Removing Neighbours

Each node maintains a set of nodes called *Neighbours* containing all of a node's neighbours. When first connecting to a node in the network, this set is empty on a new node. Therefore, the new node sends out an empty NetworkView containing only its own address. Upon receiving a NetworkView, a node calculates its local Delaunay graph including the nodes in the NetworkView. Then it notices whether it is missing a neighbour it needs to add. Every new neighbour it is already connected to is added as a neighbour directly. This might be the case if nodes are connected after bootstrapping or because one is in the routing table of the other.

If a node adds a neighbour, it has to inform its neighbours about its new neighbourhood by sending them a NetworkView. Furthermore, adding a neighbour means an existing neighbour might be removed from the neighbourhood and become an ex-neighbour. All ex-neighbours will also receive a NetworkView before disconnecting them so they know the reason for the disconnection. For example if the new node *c* lies in the middle between two neighbours *a*, *b* and *c* sends a NetworkView to *a*, *a* will add *c* as neighbour and remove *b*. By sending a NetworkView including *c* to *b* before disconnecting, *b* knows that *c* is there and needs to become its neighbour, see figure 4.7.

NetworkViews are never routed over multiple hops but only exchanged directly between connected nodes. Since the channel uses FIFO order, it is guaranteed a receiving node always acts on the most current state reported by other nodes. It will also always receive that state. Once a NetworkView is sent it will be transferred reliably. Only node fail can prevent this. However, when a node fails new NetworkViews with updated information will be exchanged anyway.

```

OnNetworkView(NetworkView nv)
{
    nodes = {LocalNode}  $\cup$  Neighbours  $\cup$  PendingNeighbours
    DelaunayGraph dg = CalculateDelaunayGraph(nodes  $\cup$  nv)
    //add or connect new neighbours
    foreach(Node neighbour in dg.GetNeighbours(LocalNode)\Neighbours)
        if(IsConnected(neighbour))
            AddNeighbour(neighbour)
        else
            ConnectNeighbour(neighbour)
    //remove pending neighbours that are no longer our neighbours
    exPendingNeighbours = PendingNeighbours\dg.GetNeighbours(LocalNode)
    PendingNeighbours.RemoveAll(exPendingNeighbours)
    //send back a NetworkView if neighbourhood views are different
    if(dg.GetNeighbours(nv.Source)  $\neq$  nv.Neighbours)
        QueueNetworkViews({nv.Source})
}

AddNeighbour(Node neighbour)
{
    Neighbours.Add(neighbour)
    DelaunayGraph dg = CalculateDelaunayGraph({LocalNode}  $\cup$  Neighbours)
    exNeighbours = Neighbours\dg.GetNeighbours(LocalNode)
    Neighbors.RemoveAll(exNeighbors)
    QueueNetworkViews(Neighbours  $\cup$  exNeighbours)
}

ConnectNeighbour(Node neighbour)
{
    PendingNeighbours.Add(neighbour)
    BeginConnect(neighbour)
}

```

Listing 4.1: NetworkView processing

```
OnConnectionOpened(Node node)
{
    if (node ∈ PendingNeighbours)
    {
        PendingNeighbours.Remove(node)
        AddNeighbour(node)
    }
}

OnConnectionClosed(Node node)
{
    if (node ∈ PendingNeighbours)
        PendingNeighbours.Remove(node)
    if (node ∈ Neighbours)
    {
        Neighbours.Remove(node)
        QueueNetworkViews(Neighbours)
    }
}
```

Listing 4.2: Connection state change

4.3.3 Pending Neighbours and Queuing of NetworkViews

However, b might not be connected to c when receiving the *NetworkView* from a . Every new neighbour a node is not connected to is added to the set *PendingNeighbours*, so b will make c a pending neighbour. Then it starts connecting to that pending neighbour. The pending neighbour is not added directly because the *NetworkView* might have travelled some time and the information within might not be up-to-date. For example, the new neighbour c might have failed in between. If a node added directly, it would probably remove some nodes that are no longer neighbours due to the new neighbour coming in. After noticing it cannot connect to the new neighbour, it would somehow need to reacquire its old neighbours. Even worse, it might have told other nodes about its changed neighbourhood before it could be sure all of its new neighbours are actually there. The information about a non-existing node could spread in the network with many nodes trying to connect to this node.

Therefore, a node only sends out information about its neighbourhood when it has no more pending neighbours. This happens when a connection to a pending neighbour was established and it became a neighbour, see listing 4.2. It also happens when connecting failed and the pending neighbour is removed without any addi-

tional action. Until then, NetworkViews are not sent out immediately but the receivers of NetworkViews are queued up. All queued NetworkView receivers get their NetworkView when there are no more pending neighbours. This way nodes only report neighbours they know they have a connection to at the time of sending the NetworkView. Of course it could happen that a neighbour failed before sending out the NetworkView but the sender did not notice this yet. But it never happens that information about a failed node is spread because every node checks the existence of the node by connecting to it first before reporting it to others.

During this phase of connecting to pending neighbours, a node still processes incoming NetworkViews as normal. Therefore, it might still add new neighbours it is already connected to, it might add new pending neighbours, or it might remove pending neighbours if it finds out another new node replaces a pending neighbour it is currently connecting to. Queuing the receivers instead of the NetworkViews gives them an up-to-date NetworkView from the time when there were no pending neighbours and not from the time of queuing it. The downside of this approach is that it slows down stabilization of the network compared to immediate reporting of unchecked information. On the other hand, it saves a lot of messages as not every single change in the neighbourhood is reported immediately but changes happening during this phase are aggregated.

4.3.4 Comparing Neighbourhood Views

When a node a receives a NetworkView from b and adds b as a neighbour, it will automatically reply with a NetworkView. If b is not a neighbour of a , then a has to decide whether to reply with a NetworkView. If every received NetworkView led to a NetworkView reply, nodes would be exchanging NetworkViews endlessly. Therefore, a compares b 's reported neighbourhood with the neighbourhood of b calculated by a using all nodes it knows. If both neighbourhoods are equal, a does not reply to b . If b 's neighbourhood is different, a replies with a NetworkView to b .

Then b calculates the neighbours it is missing and adds them or connects to them. Node b will in turn compare a 's reported neighbourhood with its own view of a 's neighbourhood. It might again queue a NetworkView for a if there is still a difference. Thus, a and b will continue to exchange NetworkViews until they finally reached an agreement on their neighbourhoods. They do not necessarily reach that agreement after just two messages. It can happen nodes keep disagreeing and ex-

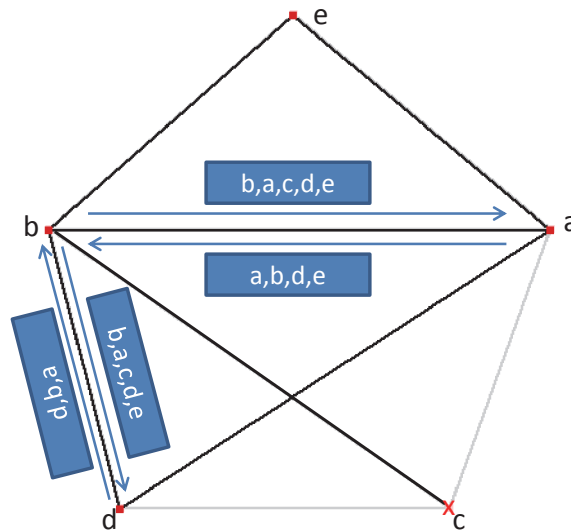


Figure 4.8: Node b detects failure of node c later than node a and node d

change NetworkViews for some time if they notice the departure of nodes at different points in time.

4.3.5 Node Departure

When a node left gracefully, it could inform all neighbours it is going to leave and perform some kind of exit procedure exchanging additional messages and handing off stored data. However, nodes are unreliable and they might also perform an ungraceful leave. If the user shuts down the computer or kills the running process, the operating system will still send out a connection close message and the node neighbours will be notified. If the user just pulls the power plug or the Internet connection fails, no notification will be sent. Only when a message is sent to the failed node, its absence will be detected after TCP acknowledgement messages timed out.

Thus, it might happen a node a detects a node failure of neighbour c really early because the connection close message was fast or a just sent a message to c and the timeout kicked in fast. Another neighbour b of c might receive the connection close message later or it might try to communicate with c later also receiving the timeout later. During that phase, a and b cannot agree on their neighbourhood views as the set of nodes they assume is existing differs.

Even worse, when b replies to a because from its point of view a is missing neighbour c , a might conclude c is a new node it has to connect to. It would add c as a

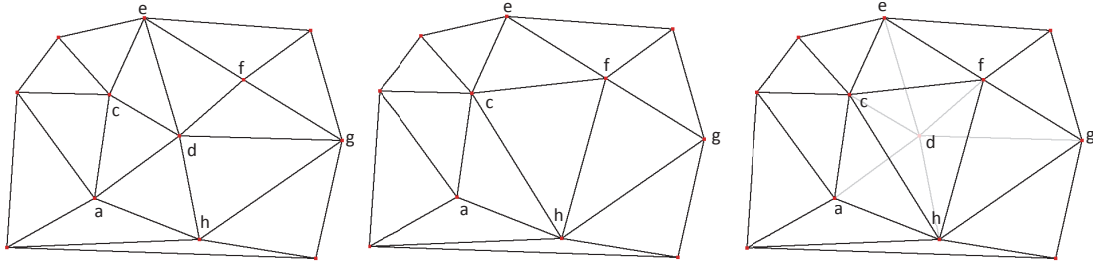


Figure 4.9: Neighbours of failed node d establish new connections

pending neighbour and only when connection establishment fails would it notice c does not exist. Until the last neighbour of a failed node actually notices it is gone, its neighbours would exchange *NetworkViews* including the failed node and try to reconnect to it, see figure 4.8. To prevent this situation, we temporarily remember failed nodes and exclude them from calculating the Delaunay graph. Furthermore, a does not reply to a *NetworkView* from b containing a neighbour he regards as failed. Node b will finally also detect the failure, change its neighbourhood, and report this change to its neighbours.

A neighbour c of node a is assumed to have failed whenever it closed the connection without first sending a *NetworkView* giving a reason for the connection close. This reason is always the existence of a better neighbour b replacing a in c 's neighbourhood. After adding this better neighbour b , c would have sent a *NetworkView* to its ex-neighbour a before closing the connection to a . Therefore, after a learned about b , both a and c would agree they are no longer neighbours. The failure of a node d is something a and c could disagree about for some time, if they detect it at different times. However, the failure of a node d is never the reason for a and c to end their neighbour relationship. When a node d fails, only new neighbour relationships are established among d 's neighbours but no relationships are cancelled, see figure 4.9. Therefore, c is always able to convince a they are no longer neighbours by sending a *NetworkView* to a . Consequently, if a was not convinced before the connection to c was closed, c must have failed.

4.3.6 Reliability and Resilience

Ultimately, the goal of a malicious node is to influence the ability of the storage to correctly answer queries. One way of doing this is by tampering with the message routing in the overlay. Since the structure of the overlay determines how messages are routed, an attacker will try to modify this structure. This allows an attacker to

permanently influence the routes messages take. If a permanent modification is not possible, an attacker might also try to cause temporary disorder in the structure to influence message routing for a certain time.

Using self-stabilizing maintenance, the structure of the system converges towards a Delaunay graph for honest nodes exchanging NetworkViews. A node not connected to its Delaunay neighbours will gradually build connections to nodes closer to its true neighbours until it connected to all of them. As long as a node exchanges NetworkViews with other honest nodes, it will get closer to its true neighbourhood. Therefore, the only chance for an attacker to prevent an honest node from reaching its correct neighbourhood is to completely eclipse [109] it from the network. An eclipsed node is only connected to malicious neighbours. All of its traffic passes through its malicious neighbours and they also control the honest node's view of the network. They will send it NetworkViews containing only other malicious nodes. The honest node will effectively be eclipsed from the correct network. Only then can the network become stable albeit with a tampered structure.

For an attacker, this is easiest to achieve when a new node joins the system. If the joining node only received malicious nodes after bootstrapping, it can be eclipsed directly. Therefore, it is crucial the bootstrapping service does not supply malicious nodes only, see section 4.4.

Once a joining node connected to other honest nodes, the success of the join process is determined by the structure of the network and the locations of malicious nodes. If at one point during the join process the joining node is connected to malicious nodes only, it can be eclipsed. Figure 4.10 shows such a scenario. In this scenario, the corridor of a joining node e to its true neighbours m , n , and d is blocked by a kind of "wall" of malicious nodes m and n . After bootstrapping e was connected to the honest nodes a and b that reported their true neighbourhood to e . Using this information, e calculates m and n to be its neighbours and connects to them. Now, n and m only report each other as possible neighbours but they exclude d which would be necessary to reach the correct network structure. Node e is now eclipsed only connected to the malicious neighbours m and n .

However, since attackers cannot chose their node positions it is unlikely there are malicious neighbouring nodes forming a wall with the necessary positions. Just a slight variation of the above scenario shown in figure 4.11 with honest node c 's position being different makes it impossible for m and n to eclipse e successfully. In this scenario, e will temporarily connect to c because it calculates c to be its neighbour together with m and n . Node c will tell e about d . Since m and n know

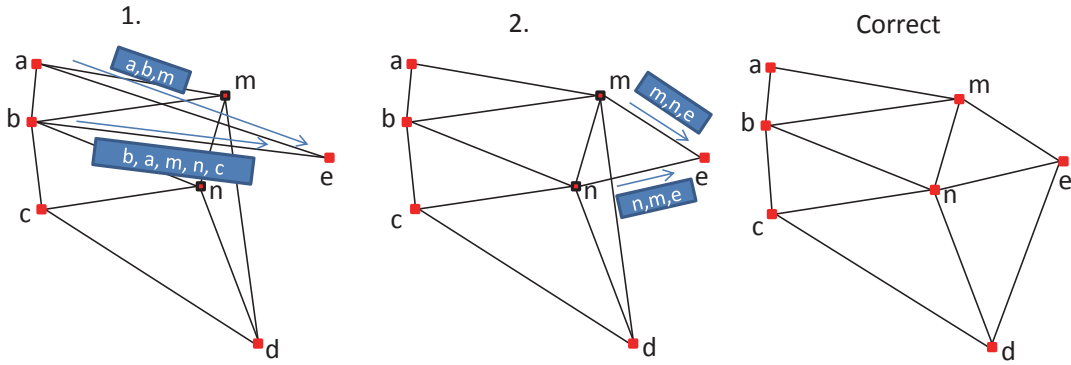


Figure 4.10: Eclipsing a joining node

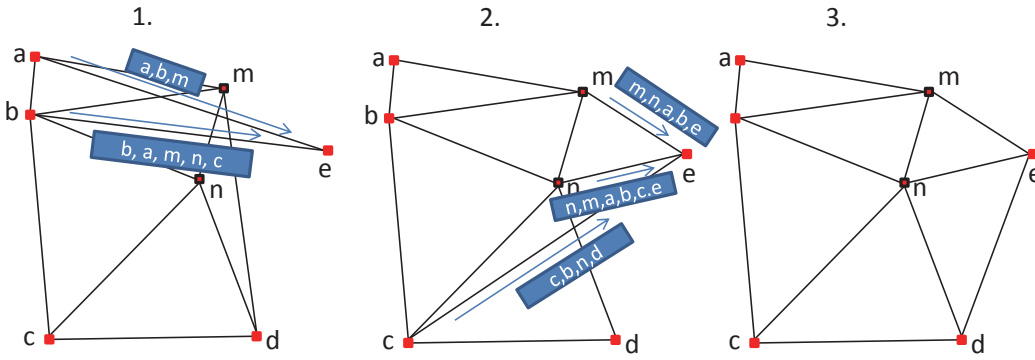


Figure 4.11: Eclipsing is not possible

this, they cannot benefit from suppressing the existence of d . Doing this would only look suspicious to e . So e will correctly join the network despite two of its neighbours being malicious. An attacker would need c as an additional malicious node to eclipse e . Having three malicious nodes this close to each other is even more unlikely due to the random distribution of nodes. Furthermore, we employed a join procedure using multiple redundant joins paths so an attacker would need suitable placement of malicious nodes on all join paths to be successful, see section 4.4.

If an attacker is not able to eclipse a node and modify the structure permanently, he might try to cause temporary disorder by sending tampered `NetworkViews` from his malicious nodes to honest nodes. If it only sent `NetworkViews` missing nodes that should be its neighbours, the receivers would just answer with `NetworkViews` containing the missing nodes. While he could use this to launch a kind of denial of service attack consuming bandwidth of honest nodes, there is no amplification. For

every tampered NetworkView sent out, one NetworkView will be returned and the consumed bandwidth is the same on both nodes.

The only way for a malicious node to cause honest nodes to change their neighbourhood is by sending them better neighbours. They would become pending neighbours on the honest node and it would try to connect to them. There is no way for an attacker to just create identities of nodes that are better neighbours for an honest node. Node identities are only created by the certification authority. However, an attacker could store the identities of nodes that have been in the network before. It could select the identities of nodes that would be better neighbours currently and put them in a NetworkView. The honest receiver would try to connect to all of these nodes. Until the connection attempts fail, it would not send out any NetworkViews and it would not be able to receive routed messages as discussed in section 4.7.

Therefore, a malicious node can temporarily influence the processing of messages in the network and timed right might be able to influence the outcome of storage queries. The only way to deal with this misbehaviour is trying to detect it. It would theoretically be possible for honest nodes to blacklist the non-existent node reported by the attacker so they would not try to connect to it for a certain time and the network stays stable. However, this mechanism could be exploited by an attacker to prevent other nodes from joining the network for a certain time. Therefore, if an honest node keeps receiving NetworkViews with non-existing nodes from some other node, it concludes that node is trying to tamper and its NetworkViews should be ignored at least for some time and this misbehaviour will be reported.

Finally, the influence gained this way is just slightly bigger than the influence gained directly modifying the results of queries received by malicious nodes as can be seen in section 5.5.

4.4 Bootstrapping and Join

With the self-stabilizing maintenance procedure, a joining node a connects to the bootstrapper node it received from the bootstrapping service assuming it is a neighbour. The bootstrapper will answer with a NetworkView and a will pick better neighbours from this NetworkView. This procedure is repeated and a connects to better neighbours until it cannot improve its neighbourhood. In the worst case, a is located in the lower right corner of the key space and its bootstrapper is in the upper right. Then a has to connect and exchange NetworkViews with all nodes in a

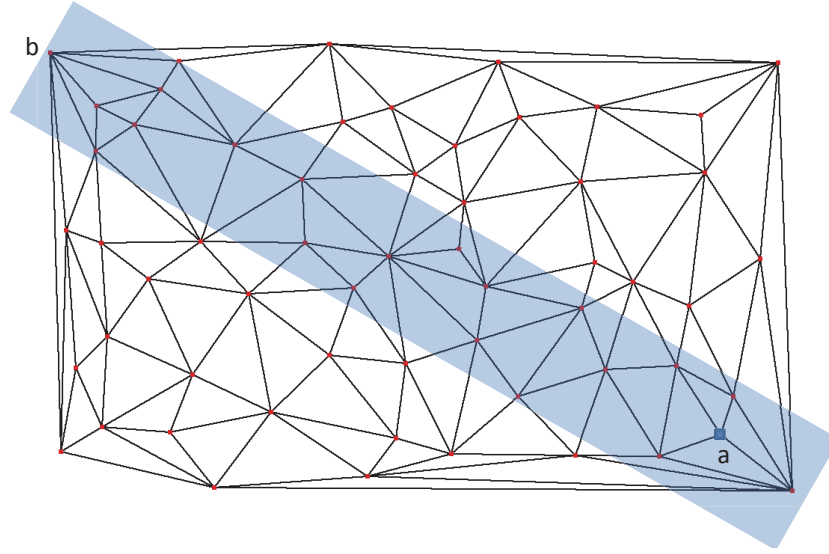


Figure 4.12: Corridor of nodes contacted by joining node a with bootstrapper b

corridor around the diagonal through the key space from upper left to lower right, while it gradually gets closer to its correct neighbours, see figure 4.12. In the worst case, $O(\sqrt{n})$ messages have to be exchanged for just one node joining. Furthermore, the more nodes involved in a join process, the more likely a constellation of malicious nodes is able to eclipse the joining node.

Therefore, we use an alternate join process first bringing a close to its correct neighbours in $O(\log n)$ steps before starting the self-stabilizing maintenance to integrate a . Instead of sending a *NetworkView* to its bootstrapper b , a sends a special *JoinRequest*. The destination of the message is a 's own position. This means the message will be received by the node c currently responsible for the position of a after $O(\log n)$ hops, see figure 4.13. This node c will definitely become a neighbour of a as no other node is closer. Node c will reply to the *JoinRequest* with a *NetworkView*, starting the self-stabilizing integration of a . As a 's new neighbours will be from c 's one- and two-hop neighbourhood, the number of nodes involved depends on the node degree (the number of neighbours). Although the node degree is only bounded by $n - 1$ for special cases like a star topology, the average node degree in a random Delaunay graph is a constant, see section 6. Thus, only a constant number of nodes are involved in this part of the joining process.

However, this procedure is more vulnerable than the self-stabilizing procedure. In the latter, a joining node is only eclipsed if the bootstrapper is malicious or malicious nodes have a special constellation in the join corridor, see section 4.3.6. Since the *JoinRequest* is routed on a single path, just one node on the path is enough to eclipse

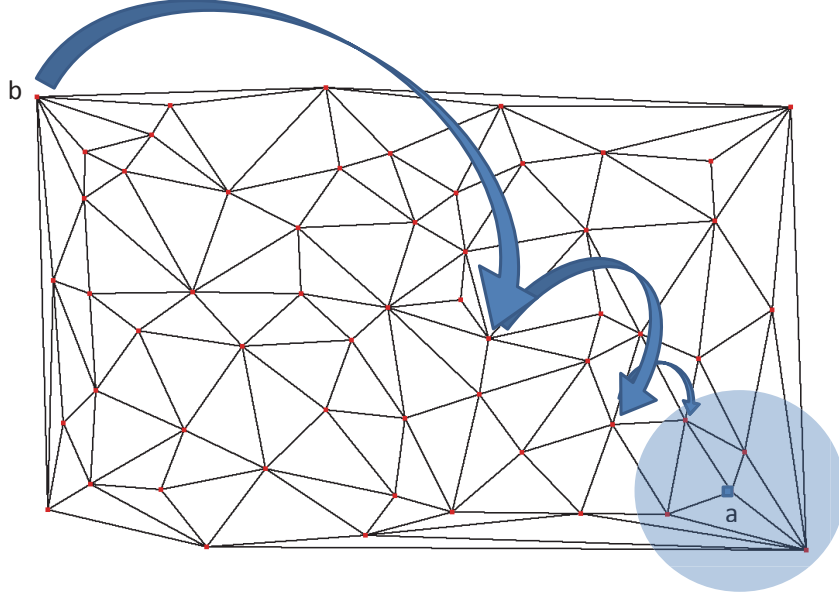


Figure 4.13: JoinRequest path from bootstrapper b to nodes involved in the join of a

a joining node. Therefore, we use multiple redundant join paths. An attacker would need a malicious node on all of the r join paths to successfully eclipse a node.

To realize these r redundant join paths, the bootstrapping service has to supply r bootstrapping nodes to a joining node. The choice of supplied nodes is a critical one. At any time, the bootstrapping service will know a subset of nodes in the network serving as bootstrapping nodes. From this subset, it will make a selection of r nodes it will supply to a bootstrapping node. This selection process must fulfil the same requirement the overlay has to fulfil: the share of malicious nodes supplied by the bootstrapping service must not be greater than the share of malicious nodes in the whole network. If the bootstrapping service is compromised and malicious nodes are delivered as bootstrapping nodes much more often, this will extend to the overlay. If it only supplies malicious nodes, every joining node will be eclipsed from the start. Therefore, a suitable bootstrapping service should effectively make a random selection from the nodes in the overlay.

Ideally, the paths taken by the JoinRequests sent to the r bootstrappers are completely disjoint. This way an attacker would need one malicious one each of the r paths. However, as all JoinRequests have the same destination, the join paths will converge at the destination. Even worse, the last node to receive the message is the same – the closest node to the destination. This means if the last node happens to be malicious, the joining node will be eclipsed despite the r paths. Therefore JoinRequests are routed slightly differently. A JoinRequest will be received by any

node who would become a neighbour of the joining node. If messages are routed towards the destination from different directions, they will end up on different nodes and the paths will most likely be disjoint. However, this cannot be guaranteed.

Therefore, we additionally take advantage of a mechanism developed to initially fill the routing table generating disjoint paths with a high probability, see section 4.5.5. This makes it hard for an attacker to prevent the JoinRequests from reaching their correct destination.

Finally, there is a second mechanism making eclipsing of nodes while joining extremely unlikely. It takes advantage of the special structure of the routing table as a result of the way its maintenance is performed (see the next section 4.5) and the fact that nodes perform a special storage query after joining the overlay (see section 5.4 and the resilience discussion in section 5.5.3).

It is important to note that initially, when the overlay is started for the first time, no malicious nodes should be present. If the very first node is already malicious, all following nodes have no chances to create a correct overlay. All of them will be eclipsed. Therefore, the operator has to supply some nodes known to be trustworthy until a critical mass of honest nodes has been reached.

4.5 Routing Table Maintenance

When a node joins the overlay, it also needs to fill its routing table. For every entry $e_{i,j,k}$ in the routing table, one node must be found matching the constraint given in section 4.2.4, which means the node must be in the corresponding zone. While the node is in the overlay, its routing table must be updated as other nodes join and leave.

An attacker will try to tweak this maintenance process in his favour. Without malicious nodes, each node will ideally have the same share of entries in other nodes' routing tables as any other node. If a node is one of 1000 in the network, it should appear in 1/1000 of all routing table entries on average. Attackers will try to let their malicious nodes get a more than proportional share of all routing table entries. When a node sends query messages for nodes to fill its routing table entries, malicious nodes will provide wrong answers putting them in place of honest nodes.

4.5.1 Constraining Routing Table Entries

An honest node will put the malicious node in the answer into its routing table as long as there is no constraint prohibiting this. Lower layers of the routing table are more tightly constrained because the zones node may come from get very small. The lower the layer, the harder it becomes for an attacker to place a malicious node there because its ID will not fit and the attacker cannot choose its IDs. However, on the top layer the constraint is not tight and any node from the corresponding top-level zone fits into an entry. Therefore, the basic idea is to put additional constraints on the nodes that may appear in a certain entry [21].

In our system, each routing table entry is associated with a so-called *target position*. Allowing only nodes within a certain distance threshold to this target position in this entry could be an additional constraint. However, this threshold is hard to pick as it depends on the number of nodes in the network. If there are only a few nodes in the network, the average distance of nodes is large. Thus, the threshold has to be larger to allow nodes in routing tables in the first place. As the number of nodes grows and the average distance shrinks, the threshold must shrink to have the desired effect of excluding malicious nodes. Global properties like the number of nodes in the system are hard to obtain. A node could estimate it using the average distance to its neighbours. However, even if a dynamic estimation of the threshold was used, a hard constraint like this would only leave the options to accept a node or to have no node at all in an entry of the routing table.

Having no node at all is not necessarily an improvement over having a node with distance above the threshold. On one hand, the probability of an entry being malicious grows with the distance to the target position. On the other hand, having no node at all means a different routing table entry has to be picked when routing. The empty entry would most likely have provided the shortest route. Thus, with hard constraints routes get longer. This again introduces higher chances of having a malicious node on the routes.

Therefore, we decided not to use hard constraints on the target positions but a preference of nodes closer to target positions. Every node fulfilling the zone constraint may appear in an entry. In the moment a node learns about a node closer to that entry's target position, it will replace that node with the closer node. Even if a malicious node manages to get into an entry by putting itself in the place of an honest node, it will be removed as soon as the closer honest node is discovered. We just have to ensure this discovery is possible.

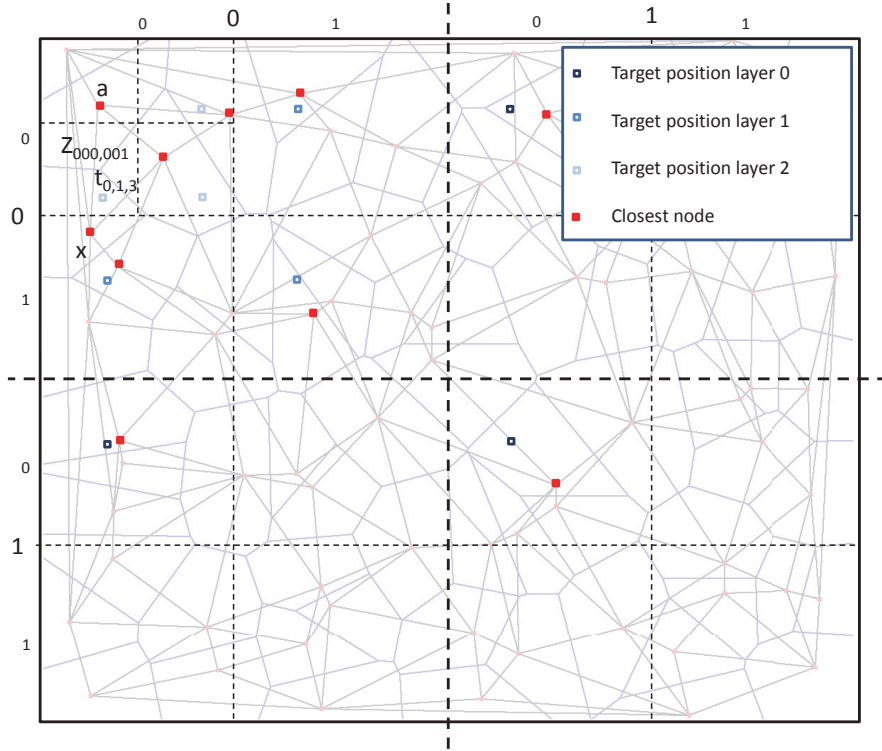


Figure 4.14: Node a with target positions and responsible nodes

4.5.2 Target Positions

The target position $t_{i,j,k}$ of a routing table entry $e_{i,j,k}$ of a node a should not just depend on the zone of that entry. This would mean all nodes use the same target position for the same zone. Consequently, the node closest to this target position will receive all traffic from nodes in other zones to this zone.

Therefore, the target position $t_{i,j,k}$ on node a also depends on the position of a . For an entry $e_{i,j,0}$ with associated foreign zone $Z_{i,j}$, the position $t_{i,j,0}$ relative to the borders of $Z_{i,j}$ is equal to the position of a relative to the borders of a 's zone. With the zones translated to be on top of each other, the target position would be identical to a 's position. The same principle is applied to lower layer entries associated with sub-zones of a 's zone. More formally, if a 's ID is $a = (a_{0,0} \dots a_{0,l-1}; a_{1,0} \dots a_{1,l-1})$, then the target position $t_{i,j,k}$ is $t_{i,j,k} = (a_{0,0} \dots a_{0,k-1} i a_{0,k+1} \dots a_{0,l}; a_{1,0} \dots a_{1,k-1} j a_{1,k+1} \dots a_{1,l})$ for $k > 0$ and $t_{i,j,k} = (i a_{0,1} \dots a_{0,l}; j a_{1,1} \dots a_{1,l})$ for $k = 0$. This means all digits are identical except for the k th digit, which is equal to i and j , respectively. Therefore, on every layer, the target position of a in another zone mirrors a 's position in its own zone

Figure 4.14 shows target positions of a node a with a routing table with $b_x = 2$ and $b_y = 2$. Thus, for each of the three layers ($l = 3$), there is one target position for

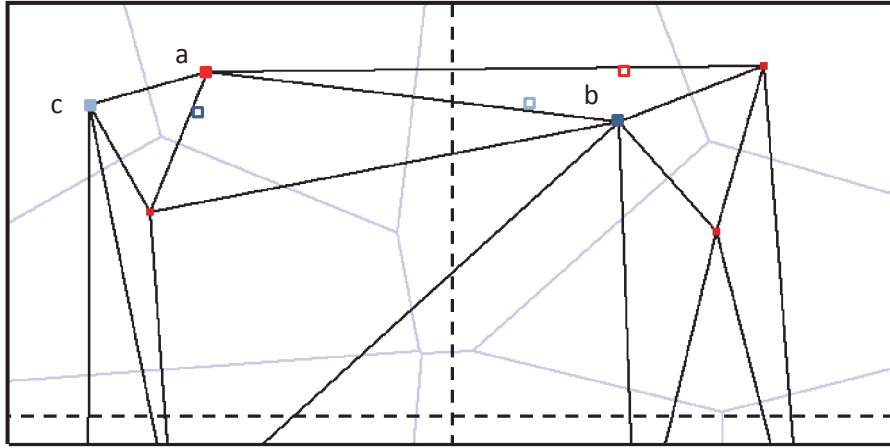


Figure 4.15: Symmetry of Routing Table Relationships

each of the $b_x * b_y - 1 = 3$ entries. The nodes closest to the target positions that will end up in a 's routing table are also shown.

Usually, the relation of one node being in the routing table of another node is not symmetric. If a node b is in the routing table of node a , this does not mean a is in b 's routing table. However, one effect of using target positions is that this relationship tends to become symmetric. If b is closest to a 's target position in b 's zone, then a is probably also closest to b 's target position in a 's zone. Figure 4.15 shows this effect. Node b is closest to a 's target position and a is closest to b 's target position. Therefore, one is in the routing table of the other. However, b is also closest to c 's target position, so b is in c 's routing table. Node b 's target position is closest to a and not to c , so a is in b 's routing table instead. Thus, the relationship is not symmetric between b and c .

This symmetry is especially useful for the passive filling of routing tables as shown in section 4.5.6 and to prevent eclipsing of node as shown in section 5.5.3.

4.5.3 Querying Routing Table Entries

To fill its empty routing table after joining, a node sends out a special routing table request message (*RtRequest*) to the target position of each routing table entry. This message is a unicast message and will be routed to the node closest to the target position. This node will answer with an *RtResponse* message and the requester will establish a connection and put this node into its routing table. However, the node closest to a target position does not necessarily fulfil the constraint of the routing table entry. It might be in a neighbouring zone.

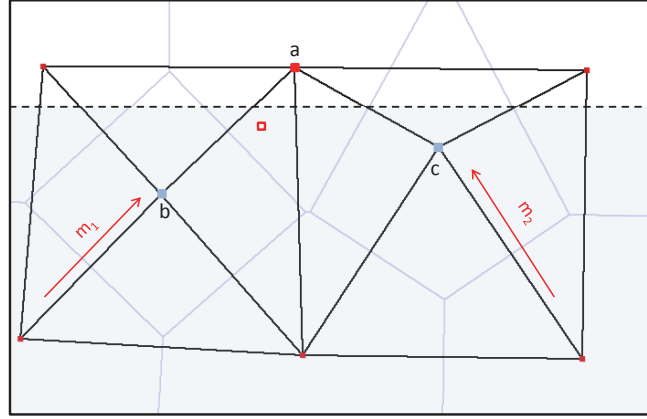


Figure 4.16: Different receivers for same destination with zone-constrained routing

This can be seen in figure 4.14. The node x responsible for target position $t_{0,1,3}$ of node a is not in sub-zone $Z_{000,001}$. It does not match a 's ID in the first two digits because it starts with $(000*, 010*)$ while a starts with $(000*, 000*)$. Therefore, it does not fit the original constraint to appear in entry $e_{0,1,3}$. If it answers with an `RtResponse`, the requester cannot add it to its routing table and the entry will stay empty.

Our first idea was to add an additional constraint to the `RtRequest` message. It should cause the closest node to a target that fulfils the given constraint to receive the message. By taking the constraint of the routing table entry the `RtRequest` was generated for, the message should arrive at the closest node within the correct zone. A node receiving such a message will forward it to a closer node as long as it does not fulfil the constraint of the message. If the receiver fulfils the constraint, it only forwards the message if it knows a closer node that also fulfils the constraint. Otherwise it is the final receiver. Therefore, once the message is in the zone defined by the constraint, it will never leave that zone and arrive at the closest node in that zone.

If no nodes on the path including the closest node are in the correct zone, then the closest node will have a neighbour in the correct zone if it shares a border with that zone. Then it can forward the message to this neighbour. The target position is always in the correct zone. The closest node might be on the other side of a zone border in a different zone. As long as there are nodes in that zone, the closest node will have a neighbour in that zone and can forward the message to reach a close node in the correct zone.

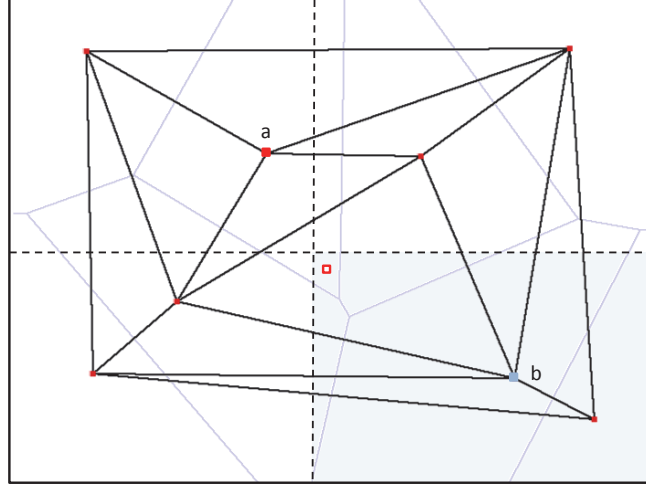


Figure 4.17: Closest node b in constrained zone is not a neighbour of closest node a

Unfortunately, this procedure does not always yield the same receiver for the same message destination. In figure 4.16, two messages m_1 and m_2 are sent to the same destination denoted by the red box. Due to the constraint, the messages should stay in the lower zone. When m_1 arrives at b , it will conclude that a is closer to the destination but a is not in the correct zone. Furthermore, b does not have a neighbour in the same zone closer to the destination so b is the receiver of the message. Node b is also globally the closest node in the correct zone so this decision is correct. At the same time, c will come to the conclusion it is the closest node because it does not know a closer node in the same zone. This decision is globally incorrect. As b and c are not Delaunay neighbours, they cannot compare their distances to agree about who is responsible. Therefore, the receiver of the message might be different depending on who sends the message and the path it takes.

Even if the routing procedure was changed to first route to the closest node and from there to the correct zone, it is not guaranteed that one of the neighbours of the closest node is in the correct zone. This can happen in the corners of a zone with two neighbouring zones. In figure 4.17, a is closest to the destination denoted by the red box. However, a is not neighbour of the node b in the correct zone, so it cannot forward the message to b .

These problems only occur rarely for nodes and target positions close to zone borders. Furthermore, getting different results is not critical for filling the routing table and the constrained routing might have been sufficient for this purpose. However, the storage also needs a way to make sure only nodes from one zone are responsible for data in that zone to realize data stored in different replication zones. This is

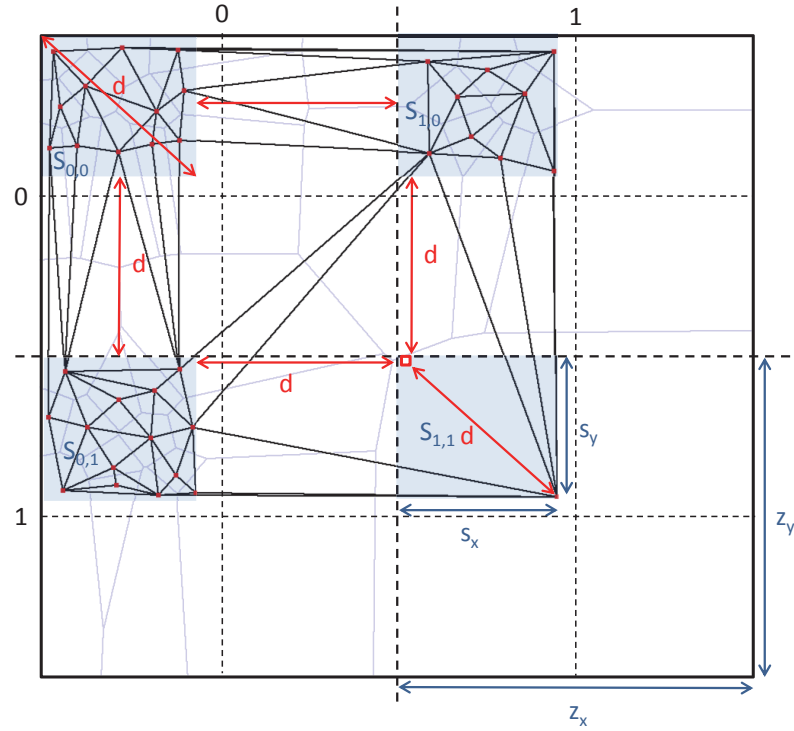


Figure 4.18: Only keys from the marked sub-areas are generated

similar to only nodes in one zone being responsible for target positions in that zone. Therefore, we only use keys from certain sub-areas of the key space to make sure the closest node to a destination is always in the same zone as the destination.

4.5.4 Key Space Usage

The aforementioned problem is created when the target position to be requested is too close to the zone border, so a node on the other side of the border is responsible for this position. If there were only target positions far enough away from the zone borders, no nodes in other zones could be responsible. Since target positions reflect node IDs in other zones, node IDs may only be taken from certain sub-areas. This is shown in figure 4.18. Only node IDs from the marked areas $S_{i,j}$ are actually generated. Therefore, all target positions are also from these sub-areas.

The diagonal d is the longest distance within a sub-area $S_{i,j}$. We picked the empty space around this area to also guarantee a minimum distance of d between different sub-areas. In the worst case shown in zone $Z_{1,1}$, the target is in the top left corner of the sub-area $S_{1,1}$ and thus closest to nodes in other zones. The only node in the same sub-area $S_{1,1}$ is in the lower right corner at maximum distance d . Even then,

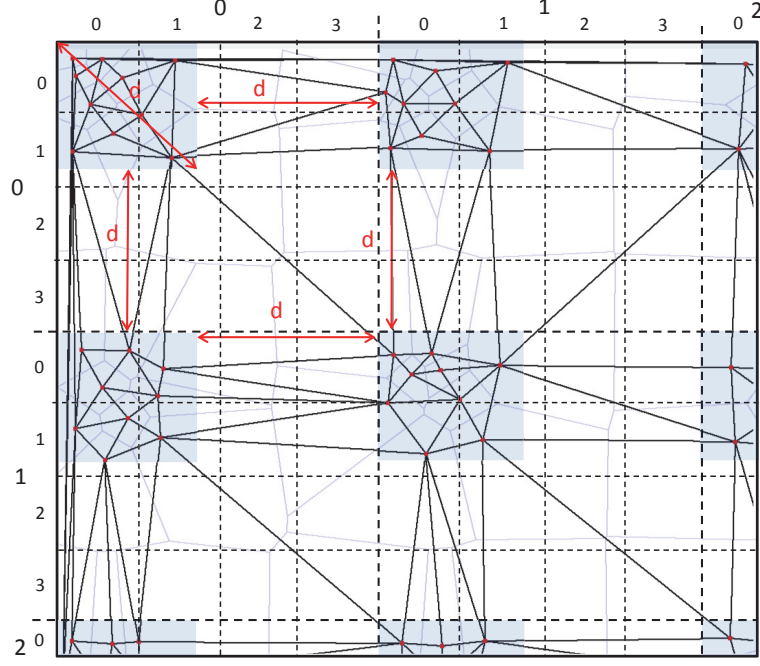


Figure 4.19: Key space sub-areas for $b_x = 4$ and $b_y = 4$

this node in the same zone is still closer to the target than any node in another zone.

The size of the sub-area s_x and s_y depends on the size of the zone z_x and z_y with $s_x = z_x - d$ and $s_y = z_y - d$. These depend on the bases used with $z_x = b_x^{l-1}$ and $z_y = b_y^{l-1}$. At the same time, the diagonal d can be calculated from $d = \sqrt{s_x^2 + s_y^2}$. Since our storage uses a square-shaped world we only consider the special case $b_x = b_y$ and $s_x = s_y$ to calculate $s = s_x = s_y$ and omit the full derivation. Then $d = \sqrt{2} * s$ and $z = z_x = z_y = s + d$. Thus, $z = s + \sqrt{2} * s$. Consequently, $s = \frac{z}{1+\sqrt{2}}$.

The empty space around the sub-area $S_{i,j}$ means the next lower level of the routing table cannot be filled completely. There are sub-zones with no nodes inside. Thus, there is no node matching the constraint of a given second-level entry. In figure 4.18, the second layer is actually completely empty for all nodes in the sub-areas as there are no nodes in all sub-zones corresponding to the second level of the routing tables. With higher b_x and b_y , there are more sub-zones with nodes, see 4.19. However, only one fourth of the entries can actually be filled. Since it is known in advance which zones will definitely be empty, their entries can be excluded from being filled by sending out RtRequests to not waste bandwidth.

We applied this scheme of using keys from sub-areas to the top-level routing table entries only. Having a node for each top-level entry is critical to realize the desired

reliability of retrieving stored data on disjoint paths, see section 5.2.2. Therefore, we had to make sure the node closest to a target position is in the correct zone on this layer. To use it on lower levels, every sub-zone currently holding keys would have to be thinned out recursively the same way the top zones have been thinned out. This creates a very uneven distribution of responsibility area sizes. Nodes at the border to the empty space would have a much larger responsibility area. On the top level, this does not matter because no data will be stored in the empty areas, see section 5.2.1. On lower levels, nodes with larger responsibility areas would have to store and maintain much more stored data than other nodes.

Therefore, we did not use this key space scheme for lower levels. Lower level entries might still be empty in case a node in another sub-zone is closer to a target position. However, this only occurs rarely and if it does, the path is not prolonged as much as if it happens on the top layer where the largest jump is made.

4.5.5 Redundant Routing Table Entry Querying

After a node a joined the overlay, it sends an `RtRequest` to all target positions of entries in its routing table. If these messages were only routed using a 's neighbours, it would mainly depend on the neighbours whether a is able to fill its routing table correctly. If the majority of neighbours happens to be malicious, the majority of `RtResponses` a receives will be from malicious nodes and a will have a malicious majority in its routing table. When another node b joins and becomes neighbour of a , a will route some of b 's `RtRequest` messages using its routing table. Thus, a will indirectly also supply malicious nodes to b although a is not malicious. One coincidental local majority of malicious nodes can easily spread out to other joining nodes.

Therefore, we would rather rely on the nodes supplied by the bootstrapping service to fill the routing table. With a high probability, it should deliver only a share of malicious nodes equalling the total share of malicious nodes in the network. In addition, we assume the bootstrapping service is aware of zones and it supplies exactly one node from each top-level zone, if existing. This way we can use these nodes to fill the top layer of the routing table initially.

When sending out `RtRequests` to the target positions of the top-level entries, each request will be forwarded to the initially supplied node currently holding the entry. When the node closest to the target position replies, it will replace that node in the routing table. However, this procedure has a flaw. It only allows our routing table

entries to get worse with respect to malicious nodes. Even if all initial nodes were honest, we could end up with a malicious entry if at least one node on one of the paths to the target positions is malicious. An initially malicious entry will definitely stay malicious.

Furthermore, this procedure relies on the one initial node from the same zone to fill all the lower layers of the routing table. If that node was malicious, all lower layer entry would become malicious, too.

Therefore, we send out multiple messages to redundantly query every routing table entry. Instead of using the normal routing procedure to send a unicast RtRequest message to the target position of an entry, we send one RtRequest for the same target position to each top-level entry. Using the standard routing, the receivers would normally forward these messages to the zone of the target position in the first step. There, several messages might travel on paths with the same nodes. If there is one malicious node on multiple paths, there would not be much benefit.

Therefore, we try to create disjoint paths to the target position of one entry by letting the messages travel in all the different zones as long as possible. We use a different kind of message that, when routed, first stays in the zone of the first hop and performs the final jump into the zone of the target position last. There, it is already at the responsible node or at least close to it so paths will be disjoint.

We realized this by using a message with an intermediate and final destination. It is first routed to the intermediate destination with the usual routing procedure. After arriving there, the final destination is put in place of the intermediate destination and the normal routing procedure continues.

For an RtRequest message, we use the target position of the requested entry as final destination. The intermediate destinations will be all the other target positions on the top layer. When one message arrives at the node closest to such an intermediate destination, and this node has a correctly filled routing table, the node will forward the message to its top layer entry for the final destination zone. There, the message will already be very close to the final destination because the receiver is closest to the intermediate node's target position.

Figure 4.20 shows the basic principle of this approach to fill one entry. Node *a* has joined the overlay. Initially, it received nodes *b*, *c*, *d*, and *e* from the bootstrapping service and put them into its routing table. These nodes are in all the different zones but they are not necessarily the closest nodes to *a*'s target position on the top layer. Then *a* starts to request entries for its routing table. In figure 4.20, the entry $e_{1,0,0}$

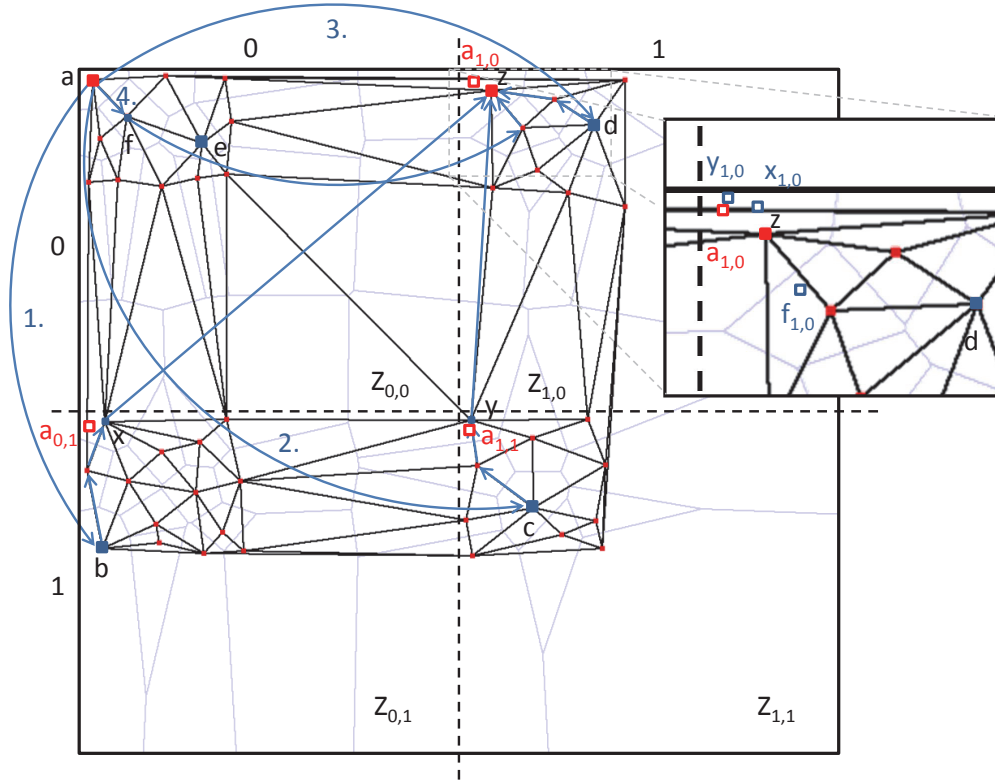


Figure 4.20: Node a requests entry $e_{1,0,0}$ with bootstrappers b, c, d, e

from the top layer of the routing table in zone $Z_{1,0}$ is requested. First, a calculates the target position $t_{1,0,0}$ in zone $Z_{1,0}$ denoted $a_{1,0}$. Then it sends out four RtRequest messages with $a_{1,0}$ as final destination and $a_{0,0}$, $a_{0,1}$, $a_{1,0}$ and $a_{1,1}$ as intermediate destinations.

The intermediate destination of the first message is the target position $a_{0,1}$ in zone $Z_{0,1}$. When a routes this message, node b from the top-level entry for this zone is closest and the message is forwarded there. Node b routes this message closer to its intermediate destination $a_{0,1}$ until it reaches node x . Node x finds out it is responsible for the intermediate destination and exchanges it with the final destination $a_{1,0}$. The node closest to this destination known by x is node z in its top routing table layer. Thus, x forwards the message to z . Since z is closest to x 's target position $x_{1,0}$ in zone $Z_{1,0}$ and x is closest to $a_{0,1}$, z is also close to $a_{1,0}$. In this example, z is even closest to $a_{1,0}$. Thus, z will reply to a with an RtResponse.

The second message is routed similarly to the first one via zone $Z_{1,1}$. Its intermediate destination is $a_{1,1}$. Node a forwards this message to c from its routing table corresponding to zone $Z_{1,1}$. This message is forwarded within that zone until it reaches node y responsible for the intermediate destination $a_{1,1}$. Node y exchanges

the intermediate destination with the final destination $a_{1,0}$. It forwards the message to node z which is in its routing table because it is closest to x 's target position $x_{1,0}$ in zone $Z_{1,0}$. Again, the message arrives at node z responsible for $a_{1,0}$ but on a disjoint path.

The third message via zone $Z_{1,0}$ is routed directly to $a_{1,0}$ because intermediate and final destination are the same and a normal unicast message can be used. Therefore, a sends this message to node d from its routing table. Node d routes the message closer to its final destination until it arrives at node z . This path is also disjoint from the other paths.

The last message via zone $Z_{0,0}$ would usually have $a_{0,0}$ which is equal to a as intermediate destination. Thus, a would receive its own message and replace this intermediate destination with the final destination $a_{1,0}$. This way the message would be routed identical to the third message. Therefore, we pick the neighbour f closest to a as intermediate destination. This is the node responsible for $a_{0,0}$ before a joined the overlay. So using it as an intermediate destination has a similar effect as before. First, the message is routed to the intermediate destination – this time in one hop directly to the neighbour. Then, the neighbour performs the jump into the final destination zone $Z_{1,0}$ passing the message to the node from the routing table closest to $f_{1,0}$. This time this is not z itself but it is a neighbour of z , so it is at least very close to z . This node will finally forward the message to z .

In this example, a will receive four answers from z . Consequently, z will definitely be put into a 's routing table. In general, this procedure creates different paths – one for each zone – to route an RtRequest to its destination. These paths are likely to be disjoint. Only in the final destination zone it might actually happen that multiple messages arrive at the same node which is not the final destination node itself but just a node close to it. However, this is not the common case. The multiple paths make it hard for an attacker to prevent a node from filling a routing table entry correctly. The attacker would need one malicious node on each of the paths, which is unlikely due to random placement of nodes.

The same procedure is executed for lower level entries. For each entry a number of RtRequest messages are sent equal to the number of zones. Each message is routed via a different zone by mapping the target position to the respective zone and using it as the intermediate destination for the message. Again, by routing to the mirrored position and jumping from there to the target position's zone, we get close to the responsible node yielding disjoint paths with a high probability that are hard to tamper with.

As mentioned in section 4.4, the same principle is applied to route the initial Join-Requests. The bootstrappers are from all the different zones making up the top level of the routing table after the node joined. Therefore, each joining node sends a message to the bootstrapper of a zone. The final destination is the node's own position but the final receiver can be any neighbour of this position. The intermediate destination is the node's top level target position of the zone of the respective bootstrapper. Therefore, the JoinRequest paths are also likely to be disjoint.

4.5.6 Active and Passive Filling

The initial filling of the routing table is done actively by the joining node sending out RtRequest messages using the procedure described before. When a node in a routing table fails, then a new node is requested using the same active filling procedure.

Using only this procedure, the set of nodes in the routing table would not change as long as no nodes fail. However, there are also new nodes joining the network that might be closer to a node's target positions. Ideally, a routing table should always contain nodes closest to the target positions for the assumptions we used before to hold. So it should be updated as new nodes join.

A straight-forward solution would be to just regularly update all entries of the routing table actively. However, a routing table has got $b_x * b_y - 1$ entries per layer. If the number of layers is l there are $l * (b_x * b_y - 1)$ entries to be requested. Each entry will be requested using $b_x * b_y$ messages on the overlay layer each of which will be routed via $O(\log(n))$ hops in the worst case. Having to transfer $l * (b_x^2 * b_y^2 - b_x * b_y)$ overlay messages on each routing table update of a node incurs a considerable overhead. For $b_x = 2, b_y = 2$, and $l = 4$ this means every node would generate 48 overlay messages. For $b_x = 4, b_y = 4, l = 4$ every node would generate 960 overlay messages on each routing table update.

Therefore, we only use an approach to passively update a node's routing table taking advantage of the active filling done by a joining node. When a node receives an RtRequest from another node, it also checks whether this node would fit in its routing table. If it is closer to an entry's target position, it replaces the current node in that entry.

This is more likely to happen the closer an RtRequest gets to the target position of an entry in one of the zones. If a joining node finds one node to be in its routing table, the joining node is likely to also end up in the routing table of the node it found or on one of the nodes close to it that were on the path of the RtRequest.

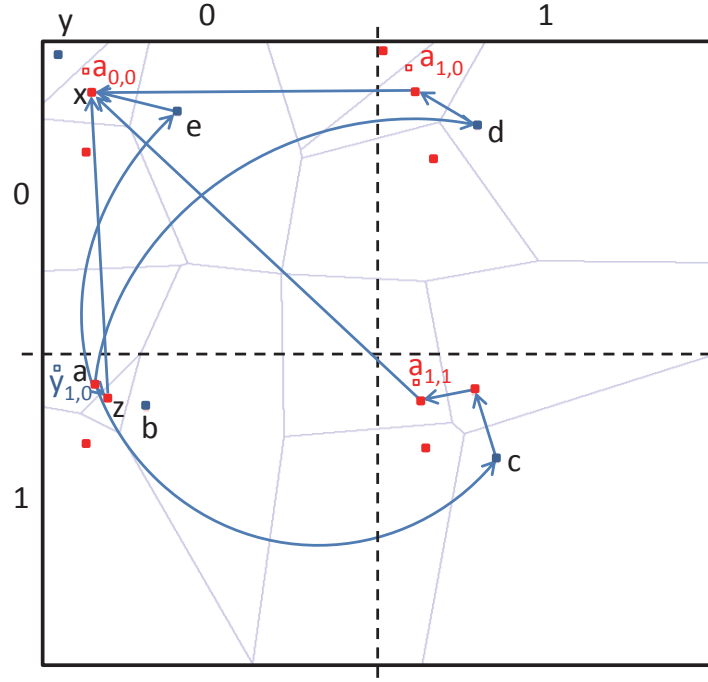


Figure 4.21: Node a requests entry $e_{0,0,0}$ with bootstrappers b, c, d, e but y does not put a into its routing table

Considering the example in figure 4.20, the nodes x , y , and z will put the joining node a in their routing tables because a is closer to their target positions than the nodes currently in their routing tables.

An attacker could only prevent z from adding a if he controlled one malicious node on all paths to z . While he could more easily prevent x and y from adding a for the shown retrieval of entry $e_{1,0,0}$, x and y will get RtRequests from a on redundant paths when the entries $e_{0,1,0}$ and $e_{1,1,0}$ are requested by a . Therefore, an attacker would need one node on each path to an entry to tamper with the passive filling.

However, since the relationship of being in the routing table of another node is not always symmetric, the passive filling does not guarantee a joining node ends up in all routing table entries where it would be closest to the target position. Such a scenario is shown in figure 4.21. Node a joins and sends out its RtRequests for entry $e_{0,0,0}$ with final destination $a_{0,0}$. It bootstrapped with nodes b, c, d, e which have been put into its routing table. Its RtRequests are routed via all different zones and arrive at the node x closest to $a_{0,0}$. Thus, a can fill its entry. However, a is closest to node y 's target position $y_{0,1}$ of entry $e_{0,1,0}$. Before a joined, z was responsible for

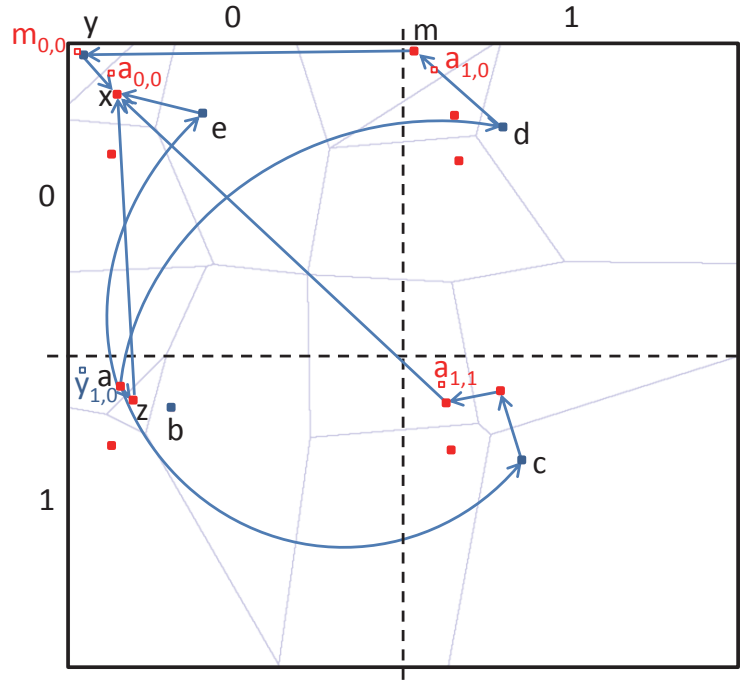


Figure 4.22: Node y passively updates its routing table after joining node a requests entry $e_{0,0,0}$

this entry. Therefore, a should replace z in the routing table of y . However, y does not receive any message from a and therefore does not know about a .

On the other hand, just a slightly different placement of the nodes in zone $Z_{1,0}$ makes node m responsible for $a_{1,0}$, see figure 4.22. Since y is closest to m 's target position $m_{0,0}$, y is in m 's routing table responsible for zone $Z_{0,0}$. Therefore, a 's RtRequest will be routed via y and y will replace node z in its routing table with node a .

Although the passive filling does not guarantee a node to always know the closest nodes to its target positions, it performs reasonably well as we will show in section 6.3.

4.5.7 Connection Management

Connections are often symmetric in the sense that both connected nodes agree they both need the connection. This is always true for neighbours. After one established a connection and NetworkViews were exchanged, they agree about being neighbours and keeping the connection open or not. Routing table connections are not always symmetric in that sense. When a node a opens a connection to a node b , then b

does not know the reason why a was interested in this connection. It might regard b as a neighbour, sending it a `NetworkView` shortly after. It might also try to add b to its routing table while b did not necessarily receive an `RtRequest` before. Node a might not have requested the entry using active filling but he could also have learned about b passively. So b does not know why a opened the connection.

However, connections also have to be closed when they are no longer needed to make room for new connections. If b is replaced by a better fitting node on a , a is no longer interested in the connection and might try to close it. However, in the meantime b might also have put a into its routing table. Thus, a may not immediately close the connection to b . Just letting the one who initiated the connection decide when to close it would not work.

There are also more complicated scenarios with interactions between neighbours management and routing table management. For example, node a and b agree they are no longer neighbours which means they could close the connection. However, one might be in the routing table of the other and might still be interested in the connection. The reason for keeping a connection open might change over time without both nodes noticing this change.

Therefore, the connection management has to keep track of the open connections of a node and the reasons for keeping it open. Furthermore, both nodes of each connection have to agree on whether it is still needed. A simple idea would be to just send a message to the other node signalling the kind of interest in the connection. However, on the receiver side there will be a delay between establishment of the connection and reception of this message. With awkward concurrent processing of messages, a node might decide to close a connection in this short time frame.

Thus, we slightly modified this idea to only send out a message when a node is no longer interested in a connection. Every node implicitly assumes that the other node it is connected to is interested in the connection and will not close it on its own. When a node is no longer interested in a connection, it sends out a *ConnectionInfo* message. When this message is received, the receiver checks whether it is interested in the connection. If it is not, it closes the connection. If it is still interested, it stores the negative interest of the other side. If at one time a node is no longer interested in a connection, it sends out a *ConnectionInfo* and closes the connection only if it has stored a negative interest of the other side. This also saves a lot of message compared to explicitly signalling the interest in a connection.

However, the network is dynamic but not all nodes notice changes at the same time because of variable message delay. One node might signal it does not need a

connection shortly thereafter noticing it does need it because of some change in the network. Meanwhile, the other node might have initiated closing the connection. In these cases, we simply reopen the connection and take care the signalling of the connection close is not misinterpreted as a failed connection attempt interpreted as a node not existing any more.

4.6 Shortcut Maintenance

The primary purpose of shortcuts is to speed up message transfer and to reduce the influence of malicious nodes on queries by sending query replies back directly instead of routing them. Whenever a node receives a routed message from another node, it creates a shortcut connection to that node, because both are likely to communicate again, e.g. because the receiver will answer the message or the sender will send a second message.

The shortcut connection is kept open for a certain time because due to the way the storage maintains the world state, the same nodes are very likely to communicate again. After receiving an answer, the requester will also establish a shortcut connection. Therefore, very often messages can be transferred with one-hop latency and queries can be answered with only two-hop latency.

Every time a message is transferred using a shortcut, that time is recorded. In fixed intervals, a node checks all its shortcuts and removes shortcuts that have not been used for message transfer for a certain time. The connection management can then choose to close the corresponding connections.

In general, using shortcuts should help to contain the influence of malicious nodes. Replies are sent directly instead of being routed. If no malicious node could tamper with the query getting to its destination on the first time, future queries will also not be tampered with. This might happen when the network structure changed and a malicious node gets on the route between source and destination. When both nodes created shortcuts to each other they will transfer all messages directly.

However, a malicious node could try to have more messages routed its way by becoming a shortcut on other nodes. It could just send messages to random destinations all over the key space to make the receivers create a shortcut connection. This definitely increases its chances of tampering. On the other hand, as long as it is not able to replace other honest nodes, it will only receive messages if it is actually the closest node. Its best chance to get more messages is to become shortcut of nodes

in other zones. When these nodes route to a node in a specific zone, they would normally pick the one node in the top level of their routing table responsible for that zone. With no other shortcuts present, the malicious node would be closer to the destination half of time on average. Thus, it would receive half of the messages that would normally be routed via the routing table entry.

This should not be a serious problem, however. First, the node will probably have other shortcuts reducing the share of messages the malicious node gets. Even more importantly, when nodes in the storage communicate with nodes in other zones to maintain the world state, they communicate with the nodes in the top level of their routing table. Therefore, an attacker can most likely not influence this most important part of the communication by creating shortcuts on other nodes. Finally, using the stored world state, the set of nodes allowed to request data can be limited as shown in section 5.5.4 so using storage-level knowledge can be used to prevent shortcuts from being established.

Another issue raised by shortcuts is the number of open connections. The number of shortcuts is normally unbounded. It might be necessary to introduce a maximum number of shortcuts because there are too many open connections that cannot be handled.

4.7 Message Routing

The overlay supports two types of messages: unicast messages that are targeted at a single position in the key space and areacast messages that are targeted at a specific area. A unicast message will be received by one node only, usually the node responsible for the destination. An areacast message will be received by multiple nodes. In our scenario, these are all nodes with a responsibility area overlapping the given destination area. It would also be possible to target nodes inside the given area only but this kind of areacast message is not employed by the storage. It would be useful for an overlay to exchange actions of players to send them to all players in range.

RtRequests and JoinRequests are a special kind of unicast messages. The only difference to a normal unicast message is that they support the exchange of intermediate and final destinations. Therefore, they are routed like two normal unicast messages. Furthermore, JoinRequests do not have a globally unique destination be-

cause any neighbour of the destination might receive the message and the receiver of the message depends on the path the message takes.

The overlay supports two geometric shapes required by the storage for areacasts: convex polygons and circles. These messages are routed like unicast messages targeting the centre of the shape first. When the responsible node is reached, a multicast tree is created to get the message to all overlapping nodes.

4.7.1 Unicast Messages

The routing process of unicast messages is straight-forward. A node compares the distances to the message destination of itself, nodes in neighbourhood, routing table, and shortcuts set and forward it to the node with minimum distance, see also section 4.2.4. The only exception to this simple procedure is when the local node is closest. Normally, the message would be passed to the application directly.

Message Buffering due to local Network Instability

However, the network could not be stable on the local node at that time. It might be joining the overlay or it heard of another joining node bound to become its neighbour. Then it would currently be connecting to some pending neighbour. This pending neighbour could be closer, but it was not included in the routing decision. If we just passed the message up to the application during that time, it might easily happen that two messages for the same destination arrive at different nodes at the same time, because they use a different set of nodes to calculate node responsibility.

Therefore, if a node is closest to a destination considering the set of known nodes but the network is locally not stable, the message is buffered. When the network becomes stable, buffered messages are routed again and either passed up to the application if the local node is still closest or they are forwarded to a closer node. We say a node is stable or unstable if its local network environment is stable or not.

To decide whether the network is locally stable, it is not sufficient to only check whether there are no pending neighbours. When a node a notices it needs to connect to a neighbour b , it will start connecting and add b to the pending neighbours set. When the connection is established, b will automatically become a neighbour. However, this does not mean a already found its final position in the network. First, a would have to send a `NetworkView` to b . When b replies with a `NetworkView`,

a might have to add additional neighbours or remove neighbours until it finds its place. Only when b does not reply with a `NetworkView` because from b 's point of view a 's neighbourhood is correct, a can consider its local network to be stable.

Defining a timeout to decide that b did not reply is difficult as it is not clear what this timeout should be. Luckily, in the above scenario b will always reply. Node a sent a `NetworkView` to b including b as a neighbour. If b does not agree with being a neighbour of a because it knows nodes that would be better neighbours, b will send a `NetworkView` back because the views did not match. If b agrees, it will add a as a neighbour and send its updated neighbourhood to all neighbours including a .

Therefore, a will always have at least one `NetworkView` of each neighbour. It can use these `NetworkViews` to find out whether its local view of the neighbourhood matches the views of its neighbours. Only when the Delaunay graph created by incorporating all `NetworkViews` from all neighbours is consistent with the local neighbourhood and all reported neighbourhoods of neighbours, only then node a considers its local network to be stable. It will pass a message to the application if it finds itself responsible. As long as `NetworkViews` of neighbours are missing or there are inconsistencies, a node remains in the unstable state and only forwards messages to other nodes, buffering messages the local node is currently responsible for.

To take that decision, every node stores the `NetworkViews` of its neighbours in a set called *NeighboursNeighbours*. When a neighbour is removed, its entry in that set is also removed. There is one special case where the `NetworkView` has to be stored although a node does not regard the source as its neighbours, see figure 4.23. If the non-neighbours a and b have a common neighbour c that fails and causes a and b to become neighbours, a and b might not detect the failure at the same time. Node a might detect the failure earlier (1.) and send a `NetworkView` to b with b being a neighbour of a (2.).

Node a already knows about b because of an optimization we used to speed up the stabilization process. Node b is a neighbour of the common neighbours c and d . Since a stored the `NetworkView` of c and d both having b in its neighbourhood, a will directly try to connect to b as a pending neighbour. This also means it will not send out `NetworkViews` only reporting the failure of c and afterwards reporting the adding of neighbour b , because `NetworkViews` are only send when there are no pending neighbours. This reduces the number of exchanged `NetworkViews` in this scenario quite substantially. After the connection to b is established, a will send its new neighbourhood to its new neighbours including b .

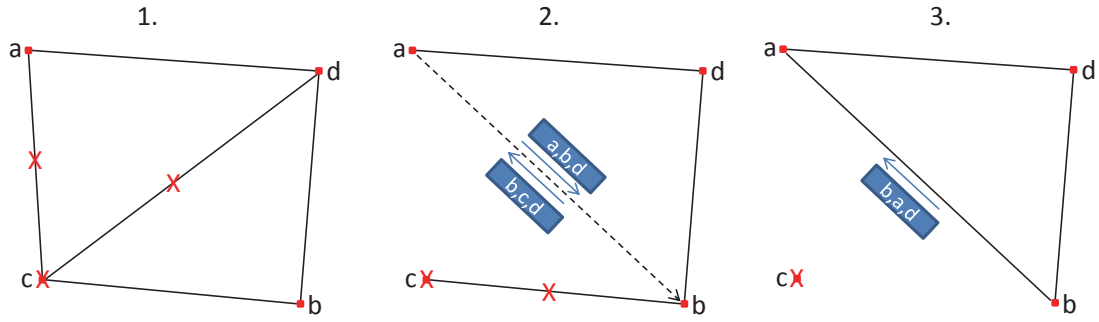


Figure 4.23: Node b detects failure of c last and a does send a NetworkView to b after becoming a neighbour of b

Since b still has c as a neighbour, it would not agree and normally not store the NetworkView of a . Instead it replies with a NetworkView including c (2.). Node b still keeps the connection open despite the neighbour relationship not being symmetric as a did not send a ConnectionInfo and b assumes a is interested in the connection, which it is.

When b also finally detects the failure, it will add a as a neighbour. After sending the updated NetworkView to a including a as neighbour (3.), a will not reply because it already added b as a neighbour before. Node a does not have to change its neighbourhood and the reported neighbourhood from b is also correct from its point of view. Finally, b would not have a stored NetworkView of a and cannot become stable. Therefore, a node also stores the NetworkView of another node if that node thinks both nodes are neighbours. Only when both nodes agree they are not neighbours, the NetworkView is not stored and stored NetworkViews are removed. When the local network becomes stable again, any buffered messages are routed again. Consequently, the buffering mechanism also introduces additional delay.

Message Buffering due to Shortcut Establishment

When a node receives a routed message, it creates a shortcut connection to the message source to prepare for possible replies to be sent back. However, the message is processed immediately and the reply message is most likely generated before the shortcut connection could be established. Therefore, messages directly targeted at shortcut nodes – typical for reply messages – are handled differently. If the shortcut connection is currently being established, the message is buffered instead of being

routed. When the shortcut connection is established, buffered messages for the shortcut node are transferred.

Establishing the connection should usually be faster than routing the message as it takes fewer messages on the link layer if the routing path is long enough. In any case, it is more reliable which is the main reason we use this procedure. However, depending on the type of Internet connection, routing the message might be faster if some expensive hole-punching procedure has to be executed to connect hosts using NAT [66]. If traffic size is not an issue, both procedures could also be executed in parallel and whichever message is first wins.

Reliability and Resilience

Routing of unicast messages is inherently unreliable. Even if we assume reliable connections, a node might always fail while processing a message before forwarding it correctly. Because of the buffering of messages, this problem is aggravated. When a node buffering a lot of messages fails, all of them are lost. With longer paths and increasing churn, the probability of a unicast message reaching its destination drops. Therefore, routing tables and shortcut sets also increase reliability of message delivery.

In addition to node failure, malicious nodes can also prevent a message from reaching its destination. They cannot tamper with the message content without detection because messages are signed but they can just drop it. The probability of a message reaching its destination considering malicious node's influence only can be estimated. Considering a routing path with an average length $\log n$ as a random sub-selection of all nodes ¹ with a malicious node share of $m, 0 \leq m \leq 1$, the message transfer will only be successful if there is no malicious node between source and destination. Thus, we can calculate the probability p all of the picked nodes are not malicious by $p = (1 - m)^{(\log_{b_x * b_y} n) - 2}$. For example for $m = 0.1, n = 1000, b_x = 2, b_y = 2$ the success probability is down to $p = 0.73$. For $m = 0.1, n = 1000, b_x = 3, b_y = 3$ the success probability is $p = 0.89$.

Another potential weakness is the fact that malicious nodes can delay the message reception on other nodes by sending them false NetworkViews. This is similar to the attack discussed in section 4.3.6 where malicious nodes provide false NetworkViews with nodes that are not in the network currently. While the reported NetworkViews

¹Which is not perfectly precise since nodes in the corner of the key space will forward fewer messages and are less likely to be picked.

are not consistent, the node will buffer all messages it is currently responsible for. However, all other messages will be forwarded as normal. Again, this kind of misbehaviour can only be detected and punished. If it reports new nodes that do not exist or it reports the failure of nodes that did not fail, it should finally be excluded from the network.

4.7.2 Areacast Messages

Areacast messages will be received by all nodes with a Voronoi cell overlapping the destination area. Convex polygons and circles are supported. When a node receives an areacast message, it checks whether its Voronoi cell overlaps the destination area. If not, the centre of the area is taken as the destination and the message is forwarded exactly like a unicast message. For simplicity, from now on we will use the term *a node overlaps an area* when we mean its Voronoi cell overlaps the area.

Multicast Tree

When the first node overlapping the destination area receives a message, the multicast handling of the message starts. A spanning tree embedded in the Delaunay graph is generated in a distributed fashion as proposed in [87]. The first node to overlap becomes the multicast tree root. This information is needed by the other nodes to calculate the tree layout. Therefore, the root node sets itself as the multicast root in the message. Afterwards, it forwards the message to all neighbours overlapping the destination area. When a node receives an areacast message with set root, it has to decide which neighbours to forward the message to. For each overlapping neighbour, it calculates how this neighbour would route a message to the root using compass routing [81]. With compass routing, the neighbour with minimum angle to the destination will be picked. The node can make that decision because it stored the neighbourhoods of all of its neighbours. Only if the neighbour would route via this node, it forwards the message to this neighbour. Otherwise another node is responsible for forwarding. Since all other neighbours of this neighbour can perform the same calculation using the same information – the neighbours and the multicast root – they will all agree who has to forward the message. Therefore, every overlapping node will receive the message exactly from one other node: its parent in the tree.

Figure 4.24 shows the multicast tree for a circular area starting at root node *a*. After *a* received the message, probably because it is in the routing table of some distant

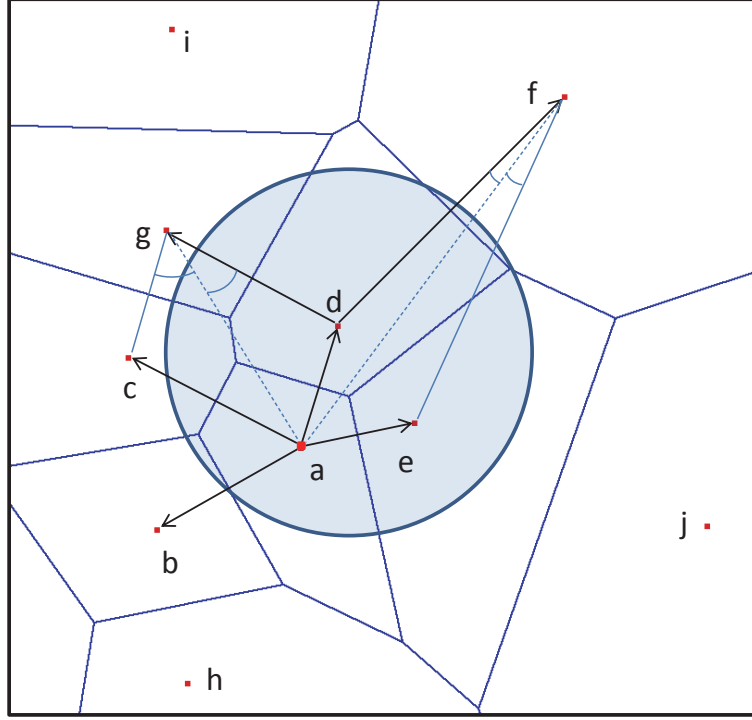


Figure 4.24: Multicast tree for a circular area starting at root node a

node, a forwards the message to its overlapping neighbours b, c, d, e . Node h is only a non-overlapping neighbour. When b receives the message, it checks whether it has to forward the message to its overlapping neighbours a and c . Node a is the root so it does not have to forward there. Node c has got the root a as neighbour so the angle from c to the root via a is zero. Therefore, b calculates c to receive the message from a and b does not forward the message anywhere.

When c receives the message from a , it also checks its overlapping neighbours a, b, d, g . Node a is the root and b and d are neighbours of a . Thus, they will receive the message from a . Node g is not a neighbour of a so c calculates how g would route a message to destination a using compass routing. Since the angle $\angle dga$ is smaller than the angle $\angle cga$, g would pick node d and c does not have to forward the message.

Node d considers neighbours a, c, e, f, g for forwarding. Nodes a, c, e are ruled out since a is the root and c and e are its neighbours. However, doing the same calculation as node c above, d finds out it is responsible for forwarding the message to g . Furthermore, when checking f , node d compares the angle $\angle dfa$ and $\angle efa$ and forwards the message to f since $\angle dfa$ is smaller. For the same reason, e does not forward the message to f . It also does not forward to overlapping neighbours a or

d. Finally, nodes f and d do not forward as their overlapping neighbours c, d and d, e are all neighbours of the multicast root a and will receive the message from a .

Reliability and Resilience

An areacast is first routed like a unicast. Until the point the multicast is actually started, it behaves exactly like a unicast reliability-wise. After that, additional nodes are involved building the multicast tree.

As long as the network does not change, the multicast tree will cover all nodes overlapping the destination area and all of them will receive the message. If nodes join or leave while the areacast is in progress, it might not work correctly. First, nodes can always fail. If a node fails while other nodes still assume it is there and forward the message according to the tree, the whole subtree of the failed node will not receive the message. The same way a malicious node can prevent its subtree from receiving the message.

Areacasts are less reliable than unicasts, as additional nodes are involved and all have to work correctly. Furthermore, it depends on the size of the destination area and the density of nodes whether an areacast is received correctly. If the key space has size s and the requested area has size a , we can estimate the success probability as follows. The number of nodes with the ability to let the unicast part of the routing fail is $\log_{b_x * b_y} n - 2$. The average number of nodes within the given area is $\frac{a}{s} * n$. However, this is just a lower bound as we are actually looking for the number of nodes overlapping the given area.

Therefore, we use an artificial number o describing the share of all nodes overlapping the requested area. Then, the total number of involved nodes is $\log_{b_x * b_y} n - 2 + o * n$ and the probability of a successful areacast is $p = (1 - m)^{(\log_{b_x * b_y} n) - 2 + o * n}$. For $m = 0.1, n = 1000, b_x = 3, b_y = 3$ and requesting an area causing a share of 0.01 of all nodes to take part in the multicast, p is down to $p = 0.31$. This means the areacast dominates the reliability with its linear dependence on the total number of nodes and the share of the requested area. Areacasts can be successful for small areas only when churn is high or malicious nodes are present.

Malicious nodes outside of the multicast tree can exert additional influence on neighbouring nodes in the multicast tree by sending them false NetworkViews similar to the tampering with unicast message routing. Thus, they can delay the processing of messages for that node and in the end they can let the area query fail as that

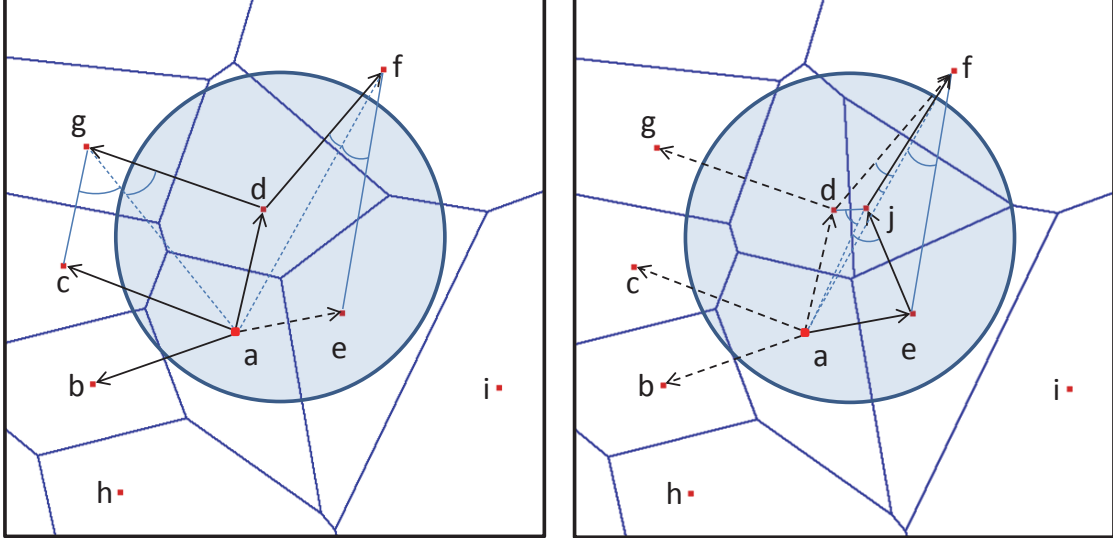


Figure 4.25: Join of node j during multicast leads to message duplication on node f

node does not answer the query in time. Again, this kind of behaviour can only be detected and should be punished by exclusion.

Duplicates

Node fail during a multicast may prevent nodes from receiving the message. However, nodes joining during a multicast alter the tree structure and can lead to nodes receiving duplicate messages.

Figure 4.25 shows such a scenario. Node a starts building the multicast tree and the message is forwarded to the nodes b, c, d, g , and f . The message forwarded to e takes longer and while being transferred, a new node j joins. During the integration, e buffers the message until j is integrated. Then, e decides which of its overlapping neighbours a, d, j, f to forward the message to. Since a is the root and d is the neighbour of the root, e does not need to forward the message to them. Comparing the angles $\angle dja$ and $\angle eja$, e concludes the latter is smaller and forwards the message to j . When considering forwarding to f , e finds out the angle $\angle jfa$ is smaller than $\angle dfa$ and $\angle efa$ so j is responsible for forwarding instead of e . For the same reason, j forwards the message to f after receiving it so f receives the same message twice – once from d and once from e .

Therefore, we use additional message IDs to remember the messages from one source already received. Every sender assigns these message IDs sequentially. However,

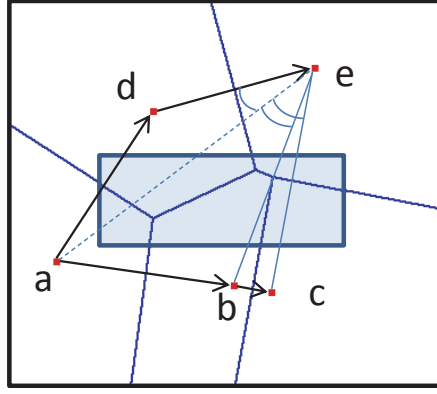


Figure 4.26: Node c cannot determine whether parent d of e overlaps and forwards

since messages can possibly overtake each other if routed on different paths due to network structure change, it is not sufficient to just remember the highest received message ID and drop any message with lower ID. Instead, all message IDs are recorded and stored for an assumed maximum message transfer time. This is done on each node receiving a message regardless of it being the final receiver. Therefore, duplicate messages are suppressed on the first node in the tree to receive a duplicate and not further down on the road when duplicates can spawn even more duplicates.

Overlap Determination

The areacast is used to reach all nodes storing data from a given area. If the Voronoi cell of a node overlaps, it receives the areacast message. During the multicast, a node has to decide whether another node overlaps. To make that decision, it actually needs to know the neighbours of that node so it can calculate the Voronoi Cell of that node. Consequently, a node can only decide overlap for its neighbours because it only stores their neighbours. However, during a multicast a node has to compare the angle of a neighbour to itself and the root to angles of other neighbours of that neighbour. When it concludes another neighbour of that neighbour has minimum angle, it will not forward the message. However, it cannot be sure this neighbour of a neighbour actually received the message because it does not know whether that neighbour's neighbour overlaps.

Figure 4.26 shows an example of such a scenario. Node c calculates node d to be the parent of e . Normally, it would not forward the message to e . However, c only knows a node d exists because it is a neighbour of its neighbour e . It does not know the neighbours of d . It cannot decide whether d actually overlaps and forwards the message. In [2], this problem is solved by c additionally querying d for

its neighbourhood. This delays the execution of the areacast by the time needed to establish a connection and to query and reply the neighbourhood. In our solution, c just forwards the message. If d also forwards, the duplicate will be removed at e . This scenario was not very common in our tests since we only request small areas with sizes similar to the sizes of Voronoi cells. Therefore, the multicast trees have a small height and most of the time nodes are close to the root. Then, neighbours of neighbours of a node are often neighbours of that node or the multicast root. Thus, a node can mostly decide whether a neighbour of a neighbour overlaps and unnecessary message duplicates are rarely generated.

4.8 Resilience

So far we have shown how some types of malicious behaviour can be prevented. Redundancy in the join and maintenance process make it very unlikely an attacker can eclipse nodes, tamper with the Delaunay structure, or get a more than proportional share in routing tables.

When it comes to the routing of messages, a malicious node can simply drop a message instead of forwarding it. Even an iterative routing procedure, where every hop connects to the source and reports the next hop with the source finally transferring the message directly to the last hop, would not be of much help. It would be very expensive. The malicious node can also always claim being responsible and answer the message. The source would have to detect there are nodes closer to the destination by sending additional messages on different routes to the destination. Instead, we could just as well send the messages on multiple paths to the same destination, like we do for routing JoinRequests and RtRequest. Thus, we do not try to detect routing misbehaviour. We use redundant paths to increase the probability a message reaches its destination.

However, there is one type of misbehaviour allowing malicious node to exert influence on message routing even if they are not on the path of the message: sending out false NetworkViews to prevent a node from processing a message when it is the final receiver. A malicious node can report the failure of an existing neighbour and cause another neighbour to buffer a message and not process it until it receives a corrected NetworkView from the malicious node and all its stored NetworkViews become consistent again, see also section 4.7.1.

Even worse, a malicious node can temporarily destabilize any other node by sending it a *NetworkView* containing a node bound to become its neighbour that is currently not online. While the receiver is trying to connect to this pending neighbour, it will not process any messages it is responsible for, see also section 4.3.6.

The key point to notice here is that the malicious node actually has to send out a (signed) message to cause the destabilization. This creates the opportunity to prove the misbehaviour to the central certification authority and punish the malicious node by revoking its certificate and exclude him from the network. If a distant node a sends a *NetworkView* containing a node c that is currently not online to a node b , a can prove this action at the CA. The CA can ask b to give a reason for sending this message to a . A possible reason would be a *NetworkView* from c that b received just recently. Therefore, time-stamps of *NetworkView* creation are included and signed in the *NetworkView*. If b is not able to give a reason, it might be excluded immediately or after the same thing happened multiple times. Therefore, destabilizing nodes by sending out additional non-existing nodes to distant nodes can be detected and punished quite easily. Consequently, this type of tampering should not occur as it is expensive to obtain new certificates.

However, the case a node reports a neighbour as failed cannot be proven so easily. A node a receiving a *NetworkView* from b missing the node c can prove b sent out this *NetworkView*. When asked by the CA, b can only justify this stating that c closed the connection – but b cannot prove this. In fact, c could have been the malicious node and on purpose closed the connection only to b to make b report a false neighbourhood change. When such a dispute is started, it is only sure that either b or c have acted maliciously. Therefore, the CA could only count the number of times certain nodes are involved in these disputes and when a node reaches a threshold depending on its lifetime it can exclude it. This could also be exploited by malicious nodes by falsely raising disputes against honest nodes using multiple malicious nodes. However, only malicious neighbours can actually send out *NetworkViews* with missing neighbours. An honest node does not care about distant nodes' reported neighbourhoods. Only when an inconsistency in the local neighbourhood is created, a node will become unstable. It is unlikely an attacker can position many malicious node as neighbours to launch such an attack. Furthermore, this will still end up being expensive for an attacker as he uses up his free disputes at the CA.

In the end, more sophisticated detection procedure correlating more factors would probably be necessary to safely identify and punish attackers destabilizing the net-

work by reporting failed neighbours. However, there is a decisive difference between the two types of sending false NetworkViews. Reporting failed neighbours is only possible to neighbours and not to other nodes. It allows malicious node to exert local influence only. Thus, nodes still have to be in the right position to tamper with specific queries. Therefore, redundant paths should still be sufficient to make successful tampering unlikely.

4.9 Discussion

The self-stabilizing overlay allows the storage to exchange unicast and areacast message to store and retrieve data. Messages are routed in a logarithmic number of steps allowing a scalable message exchange for applications. Since JoinRequests are routed to get in a logarithmic number of steps to a node's location and the self-stabilizing algorithm exchanges messages only locally, the maintenance of the primary network structure is scalable as well. Furthermore, routing tables are maintained using unicast messages which are again routed in $\log n$ steps. It also uses an approach to passively fill routing tables reducing the number of exchanged messages. However, routing table maintenance depends with $O((b_x * b_y)^2)$ on the bases of the key space adding a large constant factor to the $\log n$ steps for each message.

The overlay cannot guarantee reliable message delivery. Failure of a node means messages being processed and forwarded on that node are lost. Therefore, logarithmic path lengths reduce the chances of message loss. The self-stabilizing algorithm will repair the network structure after node fail and routing tables will be refilled by the maintenance algorithm. Thus, the application can always retry to send a lost message or it can exploit additional redundancy to compensate for the unreliable message delivery.

Finally, the overlay is resilient to attackers with a certain probability by exploiting redundancy. The primary structure cannot be tampered with without eclipsing nodes. Eclipsing nodes is hard because multiple redundant join paths have to be occupied by malicious nodes. The routing table maintenance uses redundant paths as well and the use of target positions aims to keep the share of malicious nodes in routing tables proportionally low to their total share in the network. Shortcuts allow direct replies to queries halving chances for malicious nodes to tamper with queries.

Thus, our overlay provides facilities to realize a scalable, reliable, and resilient unicast and multicast communication. A detailed evaluation of the actual results obtained can be found in section 6.3.

Chapter 5

A Virtual World Storage

As outlined in section 3.2, the task of the storage is to reliably store the state of the virtual world in a distributed fashion. Thus, players moving through the world can check whether the state they retrieved from other nodes is correct. To do so, the storage has to fulfil the following requirements.

5.1 Requirements

The main task of the storage can be divided into two subtasks. First, the state has to be stored reliably. Second, it has to be updated reliably according to the actions of players in the world. Thus, the storage has to support two main functions:

State Retrieval Nodes must be able to retrieve the state of the area they are in to check whether it is correct.

State Updating The storage must provide a mechanism to record the actions of players and correctly update the world state according to these actions.

Again, the non-functional requirements are reliability, scalability, and resilience:

Reliability Stored data must be preserved despite churn, failing nodes, and varying message latencies with late messages. The state update mechanism must record player actions and update the state reliably. Finally, nodes should always be able to retrieve stored data.

Scalability The storage must use the overlay in a way so message exchange is still able to scale to large network sizes. Therefore, data has to be stored in a distributed fashion and only a subset of nodes may be involved in operations to retrieve and update the state in a certain area.

Resilience The storage should still work reliably even if a certain share of nodes is malicious, reporting tampered state on data retrievals and tampering with the routing of messages.

In the following, we will first describe the storing of data before explaining the update mechanism.

5.2 Retrieving and Storing Data

To store data in a distributed fashion, we have to define a responsibility function mapping the position of a data item in the world to the node responsible for storing it. The self-stabilizing overlay was designed to provide such a mapping. Every node is responsible for storing all data items it is closer to than any other node. This means every node stores all data within its Voronoi cell. However, we cannot map the data space representing the area of the world directly onto the overlay ID space. Nodes near zone borders have much larger responsibility areas than nodes inside the zones creating an uneven distribution of load. Furthermore, we can take advantage of the zoning in the overlay to realize a replicated storing of data. Replication and redundancy are the main tools to provide reliability and resilience as shown in the following sections.

5.2.1 Data Space and Replication

A straight-forward way for a malicious node to prevent correct storing of data or to prevent honest nodes from retrieving the correct state is the storage and retrieval attack [111]. Instead of forwarding a store or retrieve message, a malicious node simply drops that message or replies with a faked result claiming responsibility for storing that data. One way to counter this attack is to place replicas of data in different locations in the key space so replicas can be stored and retrieved independently [59]. Since one malicious node on multiple paths to the replicas can still prevent the replica from being reached, it is beneficial to make this unlikely [5] or even guarantee paths to the different replicas to be disjoint [51].

We adapted our idea presented in [106] to retrieve replicas of data in a Pastry ring on disjoint paths to work in our two-dimensional overlay. Instead of mapping the world area directly onto the ID space, we map every position in the world to one position per overlay top-level zone. For every data item, one replica is stored in every

zone. Thus, every replica has a different position in the ID space. The replication factor $r = r_x * r_y$ equals the number of zones depending on the bases b_x, b_y of the numbering systems used.

Referring back to figure 4.18 the world area is mapped onto the marked areas $S_{i,j}$ with size $s_x * s_y$ in the zones with size $z_x * z_y$. We more formally define two linear mappings s and t . The isomorphic mapping s maps the world area

$$W = \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid 0 \leq x < w_x, 0 \leq y < w_y\}$$

with size $w_x * w_y$ onto the sub-area $S_{0,0}$: $s : W \rightarrow S_{0,0}$ with

$$s \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} w_x/s_x & 0 \\ 0 & w_y/s_y \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix}$$

Afterwards, t translates the sub-area $S_{0,0}$ and the set of replica or zone identifiers

$$R = \{(i, j) \in \mathbb{N} \times \mathbb{N} \mid 0 \leq i < r_x, 0 \leq j < r_y\}$$

to a key from the key space $K \subset \mathbb{R} \times \mathbb{R}$ using:

$$t \left(\begin{pmatrix} x \\ y \end{pmatrix}, (i, j) \right) = \begin{pmatrix} i * z_x \\ j * z_y \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix}$$

We denote the compound mapping $m = t \circ s$.

Using s , any rectangular world area can be mapped onto any rectangular shape of $S_{0,0}$. As the shape of $S_{0,0}$ depends on the zone shape depending on the chosen bases b_x and b_y , any bases can be used for the ID space.

In each zone, all nodes hold a replica of all objects in the world so there is a copy of the world in each zone. This is also shown in figure 5.1 with s mapping an object o in the world W to its replica $o_{0,0}$ in $S_{0,0}$ and t mapping $o_{0,0}$ to the replicas $o_{i,j}$ in the other zones. By sending an area query to each of the zones, every replica can be retrieved independently and majority voting can be used to decide on the correct state as shown in the following section.

5.2.2 Data Retrieval on Disjoint Paths

When a node n tries to retrieve world state, it specifies an area to retrieve all objects lying in that area. This area A can be a single point $A = p, p \in W$ or a sub-area of

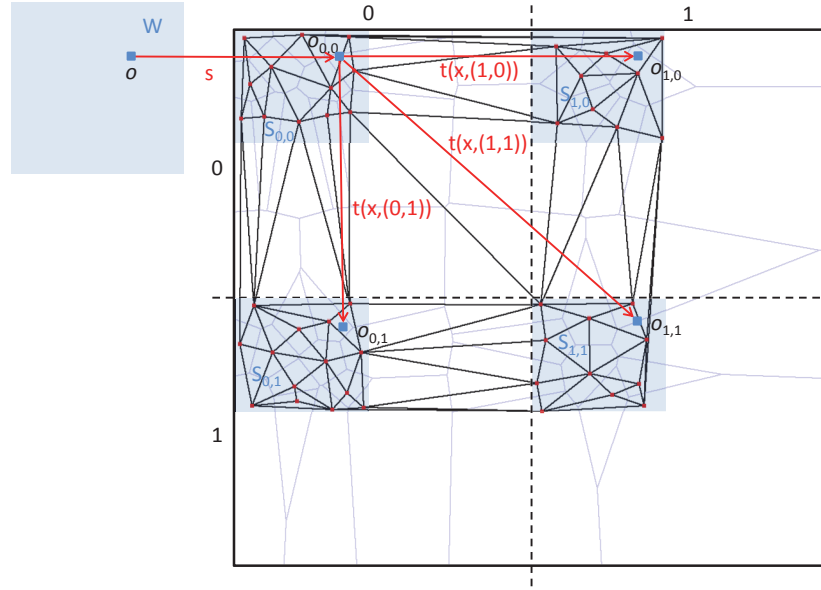


Figure 5.1: Mappings s and t map object o to its replica IDs $o_{0,0}...o_{1,1}$

the world $A \subseteq W$. The location is then mapped to the r different zones using the function m generating areas $A_{0,0}...A_{r_x-1,r_y-1}$.

Every area $A_{i,j}$ is a subset of the zone it is located in $A_{i,j} \subset Z_{i,j}$. A node only accepts retrieve operations for areas completely contained in W . Since s maps W onto $S_{0,0}$ and t just translates $S_{0,0}$ into the other sub-areas, all $A_{i,j}$ are even subsets of the sub-area $S_{i,j}$ in zone $Z_{i,j}$: $A_{i,j} \subseteq S_{i,j}$.

Sending Retrieval Messages

For each of these mapped area $A_{i,j}$, n sends a message with $A_{i,j}$ as destination using the overlay. Depending on the type of area an areacast or a unicast is used. As the overlay supports convex polygon and circle shapes only for areacasts, these are also the only area shapes supported by the storage.

The message payload is a storage message of type *RetrieveMessage* specifying the type of operation to be performed by the receiver. Furthermore, n remembers the running retrieval and starts a timer specifying a timeout.

Assuming n lies in zone $Z_{a,b}$, then the top layer of its routing table contains nodes from all other zones $Z_{i,j}, i \neq a, j \neq b$. Since all destination areas $A_{i,j}$ are in different zones, n will forward the messages to each of the nodes in its top layer. Only the message for destination $A_{a,b}$ will be forwarded to a node in zone $Z_{a,b}$ in a lower layer

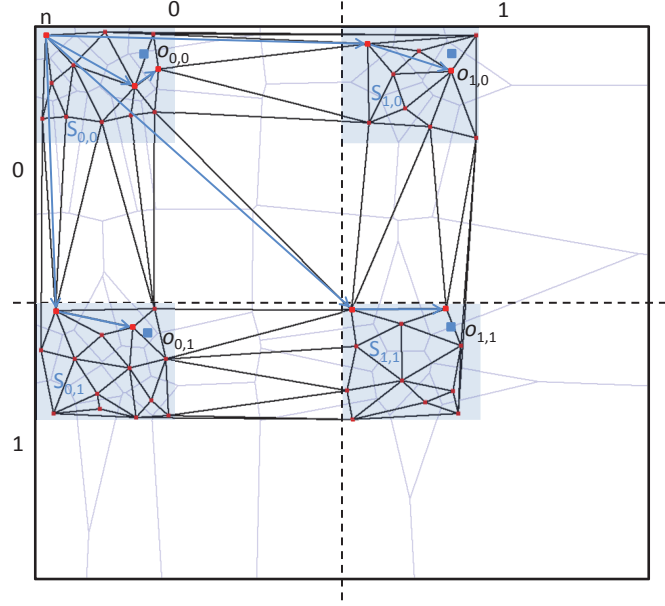


Figure 5.2: Node n retrieving all replicas of an object o creates disjoint paths

of the routing table. Therefore, in the next hop the messages will all arrive at one node in each of the different zones as shown in figure 5.2.

As long as prefix routing is used, the messages will also never leave these zones. The messages will only be forwarded to nodes matching the destination in more digits but not less. Therefore, the leading digits of all receiving nodes will always be the zone identifier of the zone the message was sent to by n . The only way for a message to leave its zone is switching from prefix-routing using routing tables to the neighbourhood-based routing.

However, the way IDs are distributed in the ID space actually prevents this. As we have shown in section 4.5.4, if there is one node in a zone $Z_{i,j}$, it is always closer to any point from $S_{i,j}$ than any node from another zone. Since every $A_{i,j} \subseteq S_{i,j}$, the nodes overlapping $A_{i,j}$ are all located in $Z_{i,j}$. Therefore, even when switching to neighbourhood-based routing, a message will not change zones.

Thus, the paths to the different replicas are completely disjoint except for node n . This contains the influence of malicious nodes because an attacker needs a malicious node on multiple paths to tamper with queries. For non-disjoint paths, an attacker only needs one common node on multiple paths to tamper.

Processing Reply Messages

When the overlay has routed the messages, all nodes overlapping the destination areas receive the `RetrieveMessage`. They select all stored replicas lying in the requested area from their local data store and answer with a reply message of type `RetrieveResultMessage`. This message will be sent directly to n using a shortcut connection the overlay will establish automatically.

Node n will record all incoming reply messages until it is able to finish the query and pass the final result back. If the query was a point-query, it will at most receive r of these messages each containing a set of replicas. When n receives such a message, it checks whether it has received more than $r/2$ messages in total. If yes, it performs a voting on the result where each message counts as one vote. If more than $r/2$ answers contain identical set of replicas, this set is returned. If not, n continues to wait for replies until r answers arrived or the timeout is reached. Then it returns the set of replicas with most votes. Furthermore, it signals its confidence in the result returning the share of votes the result achieved.

Area queries are handled in a similar way. However, for area queries the answer for one destination area $A_{i,j}$ might consist of multiple messages if multiple nodes overlap the destination area. Therefore, the replica set of a destination area is created by assembling the result sets of the individual reply messages from one zone. Furthermore, n has to decide if all reply message for a destination area have been received. This is the case when replies from all nodes overlapping $A_{i,j}$ have arrived. Since n does not necessarily know nodes in this area, this information has to be sent to n .

Therefore, all reply messages include the neighbourhood of its sender. Thus, n can check for each reply whether any neighbours of its sender also overlap the destination area. If a neighbour overlaps and its reply message has not been received yet, the replica set of this zone is still incomplete and n will wait for further replies. Only when more than $r/2$ replica sets have been received, the above voting procedure starts with each replica set having one vote. It returns, when a majority is reached or waits for further replies until the majority or the programmed timeout is reached.

5.2.3 Storing Data

The storage does not only allow retrieving static state. It also updates the world state. Consequently, the storage also allows storing of information. When a node

wants to store an object in the world, it creates r replicas of this object and sends a unicast *StoreMessage* containing the replica to the object position mapped to the different zones. Nodes receiving such a message add the replica to their local data store. Since storing and retrieving data both use the same mapping and the same routing procedure, the message paths for storing data are again completely disjoint.

Storing data is a fire-and-forget operation. No acknowledgement messages are sent back. Replication and disjoint paths should be sufficient to have data stored at the majority of responsible nodes. An alternative would have been to use r different paths to each of the r replicas, similar to the routing table maintenance performed by the overlay. However, storing data is a much more common operation than node join and leave and the traffic overhead would probably be prohibitive.

5.3 World State Updating

A central server can update the world state directly after receiving an action from a player. It can decide which actions are valid, calculate their effects, and apply them to the current world state closely after the action has been issued by a player. These updates happen continuously at a fast rate and the world state is always up-to-date at the server. Only the client views experience a small delay displaying the world state.

Unfortunately, realizing this in a peer-to-peer system is impossible for two reasons: distribution of state and replication. The former is necessary to make it scalable. The latter makes it reliable and resilient. In the following section, we will analyse why state distribution and replication prohibit fast-paced continuous updating of the world state.

5.3.1 State Distribution and Replication

In contrast to a central server, a node a is only responsible for storing and updating a sub-area A of the world. This area contains the objects $O_A = \{o_1, \dots, o_n\}$ and t_0 is the timestamp of these objects – the world time they have been updated last. Node a then updates these objects to time t_1 . Possible reasons include the arrival of a player action with timestamp t_1 or a continuous update is just scheduled to happen. Using an event-based simulation, a needs all relevant actions of players that have

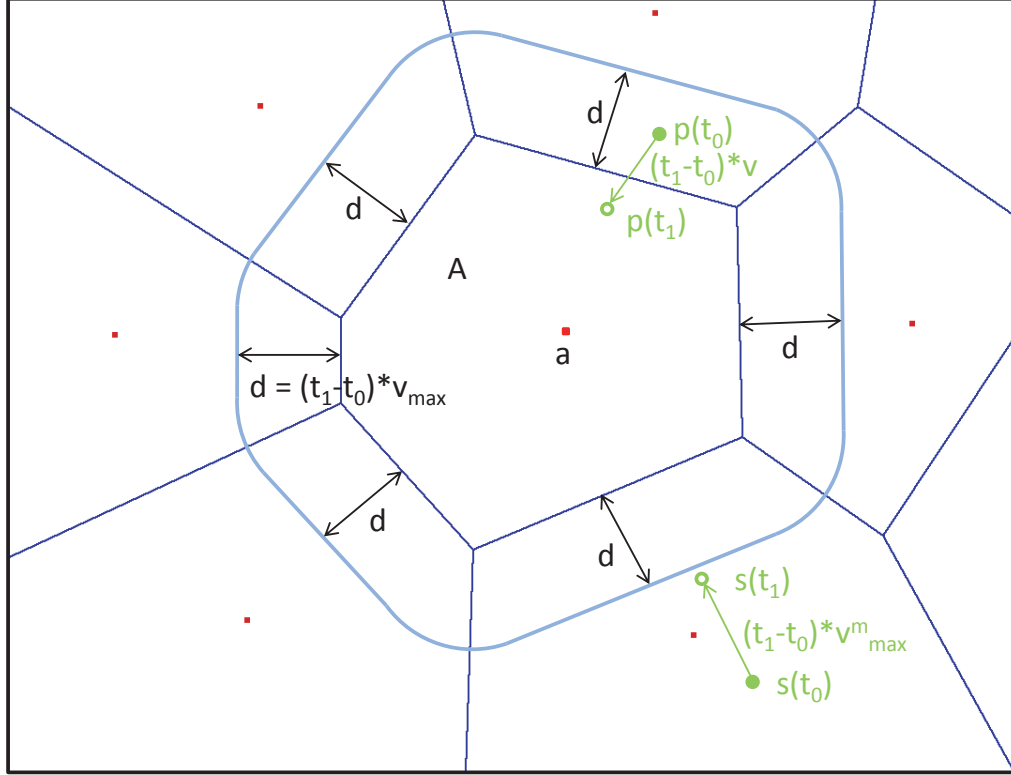


Figure 5.3: Player p outside of A at time t_0 moves to arrive inside A at t_1

happened since the last update time t_0 to calculate the new state of all objects in O_A .

Movement

Our main finding is that only including the objects in O_A and the actions of all player characters in O_A is not sufficient to correctly update the state. A player character p could be outside of A at t_0 but moving into the direction of A at speed v with the result of being inside of A at t_1 . Node a needs the old state of p at t_0 including its movement vector to calculate its position inside of A at t_1 . This scenario is shown in figure 5.3.

On the other hand, p could also have stopped moving shortly after t_0 so he will not be inside of A at t_1 . To correctly reflect this, a does not only need the state of the outside object p at t_0 , it also needs all movement actions generated by p between t_0 and t_1 .

Finally, p could also have an indirect influence on objects inside of A . He could be standing still just outside of A and an AI-controlled character executes an AI-update

event between t_0 and t_1 . The AI performed some random wandering behaviour looking for players to chase and happened to move close to p . Executing the event, the AI would inspect its environment and notice the player being inside its perception range and would start to chase the player. Therefore, even if the player character outside of A performs no actions and does not move, its mere presence can influence the correct updated state.

However, not any object in the world can influence objects of A . A player character can be sufficiently far away so he cannot move into the area A between t_0 and t_1 as shown for player s in figure 5.3 moving towards A with speed v_{max}^m . More precisely, he could not be close enough to move into A or into the perception range d_{AI} of any AI inside A . Then, he cannot influence the state of objects in A at time t_1 . This distance threshold d depends on two parameters: the maximum movement speed v_{max}^m of mobile objects and the time passed between t_0 and t_1 : $d = v_{max}^m * (t_1 - t_0) + d_{AI}$.

Interactions

Outside player characters or AIs can also interact with objects inside A by performing actions. Fortunately, similar to the real world, actions only have local effect. Throwing a stone at somebody or casting a spell can only affect objects in a certain maximum distance to the object performing the action. However, just adding the maximum interaction range d_i of any action to the above formula is not sufficient.

For example, player p_3 outside A triggers an action to throw a stone at player p inside A at time t_{p3} between t_0 and t_1 and they are exactly d_i distance away from each other. Then, depending how fast the stone flies, it will hit p at some time between t_{p3} and t_1 and will have an influence on the state of p at t_1 . Now a third player p_2 behind p_3 from p 's perspective at distance d_i from p_3 throws a stone at p_2 at time t_{p2} between t_0 and t_{p3} . If this stone hits p_3 before t_{p3} , it might have an effect on p_3 's ability to throw stones. It could reduce p_3 's chances to hit p and p_3 's stone misses p leading to a different state of p at t_1 . Thus, actions can also have transitive effects on objects in the world.

Whether such a transitive influence is possible depends on the maximum propagation speed of interaction effects v_{max}^i . If performing an action with effect at distance d_i only takes time x with $t_1 = t_0 + 3x$, this interaction can happen 3 times between t_0 and t_1 as shown in figure 5.4 giving a scenario of chained interactions from player

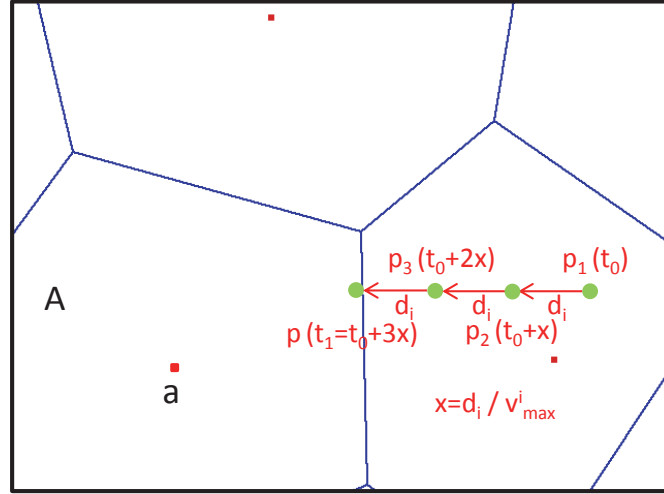


Figure 5.4: Interaction chain with transitive effect from player p_1 to p

p_1 to p_2 , p_3 , and p . In this scenario, everything closer to A than $3 * d_i - p_1$, p_2 , and p_3 – has to be included to update the state in area A at t_1 .

The maximum influence distance from A can be computed from $d = v_{max}^i * (t_1 - t_0) + d_i$. The first part is the maximum transitive range of action effects. The maximum interaction distance d_i has to be added because it decides whether an player outside A is able to start an interaction with any object inside A . Every node has to maintain running actions affecting objects inside its area. Suppose action effects propagate really slow but at a long distance d_i . This means it would take long until a triggered action has an effect on an object inside A . Without adding d_i in the formula, a would simply miss the triggering of the action by the far away player character and would later not be able to update the state of the action target correctly.

The interaction speed is usually much higher than the movement speed as a thrown stone usually flies faster than any player can run. The maximum interaction range of characters is also mostly higher than the AI perception range. Therefore, the above interaction formula dictates the additional maximum influence distance around A to calculate the new state of objects in A .

Effects of Distribution

This dependence of the updated state from an increased area does not allow fast continuous updating of the state. Before any incoming event can be processed, the

old state of the additional area has to be fetched from some other node updating that particular state.

It is very unlikely a player at distance d moves towards A on the shortest path so he ends up in A or that a chain of player interactions builds up from maximum range in a way that they influence objects in A . However, a does not know this when it starts updating the state at t_1 . It has to include all additional objects and player actions only because they could have an influence to always calculate the correct state. Most of the fetched objects will probably not end up in A and can be deleted after updating. Node a would not be able to calculate the correct state of objects outside of A anyway as it would need all objects inside distance d from these outside objects. Updating these objects is the task of other nodes.

This mechanism actually depends on the game rules to realize a scalable solution. Player actions with instantaneous effects on distant locations must not exist. Such an action would mean a needs to fetch the whole world state as any player in the world could have indirect effect on objects in A , preventing a scalable solution. From a scalability point of view, low movement speed and low interaction propagation speed would be best as they reduce the number of objects to be fetched.

Allowing only interactions with slow speed instead of fast-paced interactions would probably be boring for players. However, actions like shooting an arrow could always be realized by first adding a cast or aiming time after the action has been triggered. When this time is over, the arrow flies at the expected high speed. The propagation speed from triggering the action to the arrow hitting its target is slower, depending on the aiming time.

Effects of Replication

Nodes can leave the network unexpectedly at any time. To prevent the loss of data, stored data has to be replicated onto multiple nodes. These replicas have to be consistent so nodes can retrieve the correct world state. The state of an object in the world depends on the actions of players. Due to varying message latencies, these actions can never arrive at the same time or in the same order on all nodes holding a replica of an object.

Optimistic simulation techniques using rollbacks can be used to realize eventual consistency. Thus, replicas would become consistent when no more actions are generated. However, actions will always be generated as long as players are in the world so there would never be a point in time where replicas are actually consistent. This

means voting procedures cannot be used to compensate for tampering by malicious nodes as replicas would have different values anyway. Thus, fast continuous updating of the world state with immediate action processing is impossible when a state of actual replica consistency should be reached. Then, voting can be used to make the storage reliable and resilient.

5.3.2 Update Area Retrieval

As we have shown, processing actions immediately to update the state is impossible when objects are distributed among the nodes. Therefore, nodes have to update their object states at specific times t_0, t_1, \dots . These times are the same for all nodes so updates will be synchronized and all replicas of an object will have the same timestamp when the update is completed. This allows performing a majority voting to obtain the correct state with a high probability. Updates happen in fixed intervals with length $I = t_{i+1} - t_i, i \in \mathbb{N}$.

However, the latest state in the storage will always lag behind the current time in the virtual world. Nodes trying to obtain the correct state around the current position of their player characters at time t can only do so for the latest update time t_i with $t - t_i = \min, t - t_i \geq 0$. If it happens to request the state shortly before the next update, the storage can only return a state with age nearly I . However, we will show this state is still useful as it can be updated to the current time.

The update process is based on our analysis of distributed state updating in section 5.3.1. Suppose node a is responsible for maintaining a sub-area A of the world. This area A can be calculated by using the reverse mapping m^{-1} on a 's Voronoi cell in the ID space transforming it into the world area. Now, a wants to update the state of objects in its area A from time t_i to time t_{i+1} . To calculate the updated state, it needs the state of an increased area around A from time t_i . This area depends on A and the maximum influence distance d around A according to the formula $d = v_{max}^i * (t_{i+1} - t_i) + d_i$, with v_{max}^i being the maximum interaction propagation speed and d_i the maximum interaction range. The shape of the resulting area resembles the polygon A but has rounded corners as shown in figure 5.3.

Node a has to fetch this area by performing a retrieve operation on the storage. Basing on the overlay, the storage only supports retrieve operation for circle and polygon shapes. While it would be possible to calculate area overlap between polygons and the created more unusual shape, we opted to use simpler method by retrieving the minimum enclosing circle covering this area instead. As shown in figure 5.5, a first

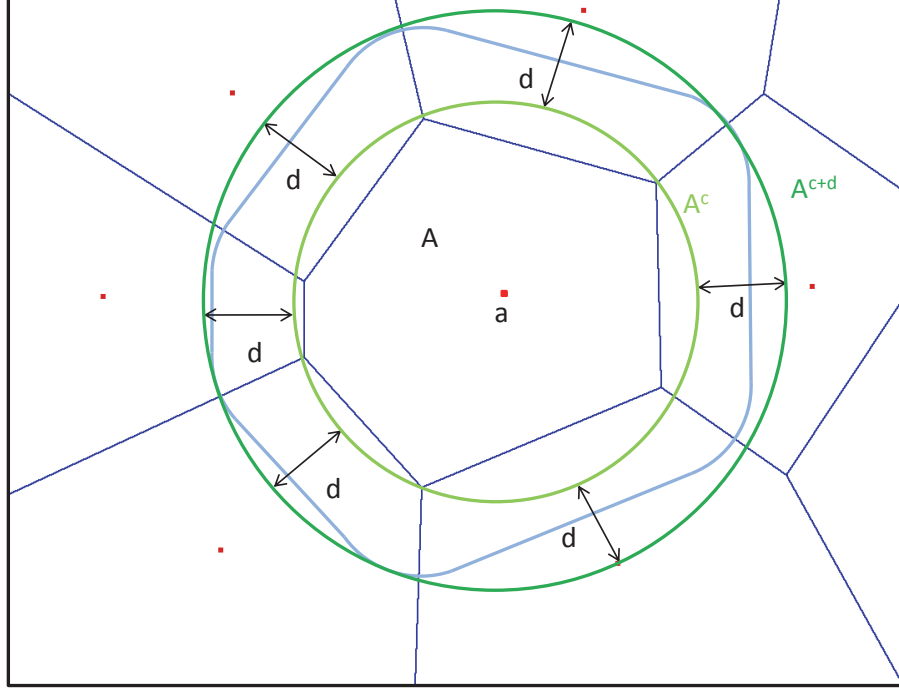


Figure 5.5: Circle A^c covering A with increased circle A^{c+d}

calculates the minimum enclosing circle A^c around A and then adds d to its radius to obtain the requested circle A^{c+d} . We call A^{c+d} the *update area* as it contains the objects a uses to calculate the updated state.

The storage executes the retrieval operation, fetching the replicas and performing a majority voting to return the state of objects in A^{c+d} at time t_i . The result is the set of objects that may have an influence on the state of objects in A at t_{i+1} and that could be inside of A at t_{i+1} . Now, a only needs the actions of all players inside A^{c+d} between time t_i and t_{i+1} to calculate the updated state.

5.3.3 Player Actions

The actions performed by players determine the state of the world. Starting from the initial state, it is always possible to calculate any updated state if all player actions are available. Node a needs all actions of players inside the update area A^{c+d} between time t_i and t_{i+1} to calculate the state of A at t_{i+1} so it has to obtain them somehow.

Since player actions are so decisive for updates and we have to ensure nodes can agree on the performed actions, we also use the storage to store these actions.

Whenever a player performs an action, it also performs a store operation on the storage. The position of the stored action in the world is the ID of the storing node mapped into the world area using m^{-1} . This means a node stores the actions of its player character together with the nodes in other zones storing their replicas. These nodes are the authority on the actions performed by a player. Similar to the voting approaches in NEO [48] and SEA [30], only when the majority of these nodes returns the same actions, these actions will be considered as being triggered by the player. If a player wants to perform an action in the world, it must ensure this action is actually stored. This principle also allows dealing with timing cheats since the player has to commit itself by a certain time by storing its actions. Store messages arriving late at the storing nodes will be rejected.

To not mix up world objects with stored actions, we introduced different storage containers that can be addressed in the store operation: the world container storing world objects and the action container storing player actions only. Nodes only accept store operation for the action container to store player actions. Writing to the world container is not possible. A node only changes the data in its world container during the update process but not by executing store operations from other nodes, so state cannot be modified by other nodes directly.

When a obtained the state of objects inside the update area A^{c+d} , it knows the IDs of all player characters in that area. It retrieves all actions of these player characters between t_i and t_{i+1} . Finally, it can compute the correct state of objects in area A at time t_{i+1} .

5.3.4 Circular Maintenance Area

Node a can actually compute the correct state in the minimum enclosing circle area A^c around its polygon A . Since a retrieved the data and calculated its update, a does not delete this data but keeps it after the update. Therefore, the area of the world maintained by a is actually the larger circle area A^c . We call this area the *maintenance area*. This area overlaps with the areas of a 's neighbours and the objects in the overlapping areas are maintained on multiple nodes.

Upon retrieval, nodes will still return all objects in the requested area they are maintaining. Thus, duplicates of objects might be created when the requesting node assembles the result from one zone. If different versions of the same object exist or one result message is missing an object which is contained in another result message – which could only happen if malicious nodes tampered with object state

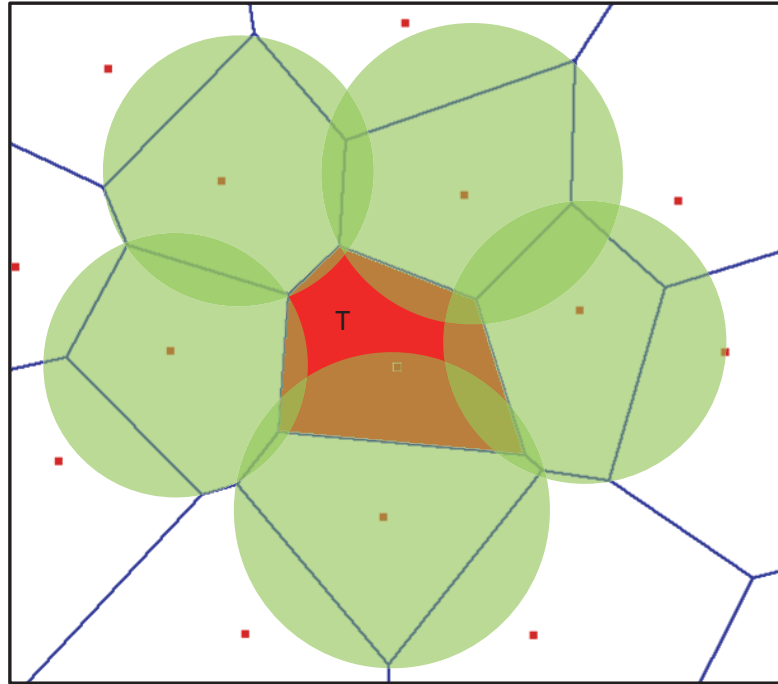


Figure 5.6: A malicious node can only tamper in the sub-area T of its responsibility area due to overlap with honest neighbours

or created/dropped objects – voting is used. If one object version has a majority, this one will be returned. If votes are equal, the complete result from the area is discarded. Chances at least one of multiple objects with conflicting version is not correct are very high so completely discarding the state and relying on better results in other zones is a better alternative.

This mechanism makes it harder for malicious nodes to tamper with data. If it tampers with objects in areas overlapping with its neighbours' maintenance areas, the result will not be counted. To successfully modify an object, it can only try to get a majority in areas not overlapping honest nodes' areas. Figure 5.6 shows how a malicious node can only try to tamper successfully in the area T not overlapping the maintenance areas of its honest neighbours. However, this mechanism also increases sizes of retrieval return messages as more objects are contained.

5.3.5 Update Process Timing

The state retrieval and action retrieval operations have to be timed right so all nodes synchronize updating their world state. Suppose the updated state of time t_i has just been calculated and nodes are preparing for the next update to time t_{i+1} . After

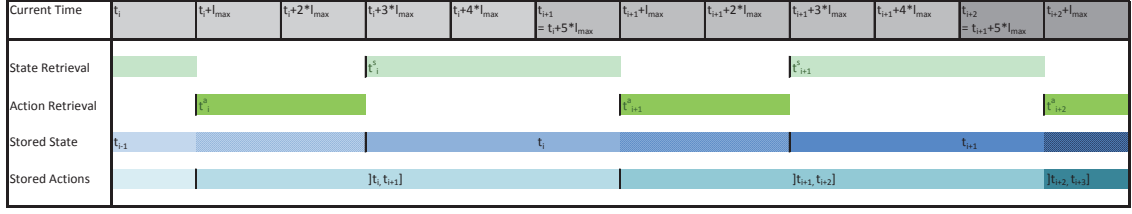


Figure 5.7: Timing of retrieve operations in the update process

a node a updated its state to t_i , it has to retrieve the update area A^{c+d} from the storage to obtain the set of objects able to influence A^c . However, before it can do this it has to be sure the other nodes also successfully updated their state to time t_i . We denote this time t_i^s with $t_i < t_i^s < t_{i+1}$ and explain its calculation later. At t_i^s nodes start to retrieve the state of time t_i .

When this retrieve returns, nodes have to retrieve all actions of players in the retrieved update area A^{c+d} between t_i and t_{i+1} . Before nodes can request all actions including t_{i+1} , they have to make sure these actions have been stored successfully. At latest, a player could generate a relevant action at t_{i+1} . After generating, this action still needs some time until all of its replicas are stored at the different nodes.

Since a player generating actions always stores these actions on the same nodes, the overlay will have established shortcut connections to these nodes. Storing a replica just takes the latency of one message on the link layer. We define l_{max} to be an assumed maximum link layer message latency. All assumptions about whether nodes have completed phases of the update process are based on this latency. Most of the time, this assumption will hold. If it does not and a message is late, a node will report wrong data. Replication corrects for this and ensures the updating works correctly with a high probability.

Figure 5.7 shows the timing of the update process based on the assumed l_{max} . To calculate the state at time t_{i+1} , a node needs all actions of players generated until this time. As l_{max} is the maximum duration of store operations, all actions up to time t_{i+1} should be stored at $t_{i+1}^a = t_{i+1} + l_{max}$. At t_{i+1}^a , nodes retrieve the actions of players in their update area between time t_i and t_{i+1} . We assume $2 * l_{max}$ as bound for the duration of the retrieve operation as it consists of one retrieve message to the storing node and one result message back to the retrieving node. Consequently, $2 * l_{max}$ is the timeout for action retrieves. Therefore, at time $t_{i+1} + 3 * l_{max}$ all nodes should have completed their retrieve operations for all actions up to $t_i + 1$. Therefore, they can calculate the state in their maintenance area A^c at time t_{i+1} . They actually perform that calculation upon retrieving the last missing

player actions so the state updates at the different nodes will happen somewhere between $t_{i+1} + l_{max}$ and $t_{i+1} + 3 * l_{max}$.

At $t_{i+1}^s = t_{i+1} + 3 * l_{max}$, nodes can be sure all other nodes finished their update to time t_{i+1} . Thus, nodes can start retrieving the state at t_{i+1} to prepare for calculating the state at t_{i+2} and the update cycle starts again. The cycle length is $I = 5 * l_{max}$, yielding $t_{i+1} - t_i = 5 * l_{max}$. Consequently, the aforementioned t_i^s can be calculated from $t_i^s = t_i + 3 * l_{max}$.

The operations *State Retrieval* and *Action Retrieval* shown in figure 5.7 run interchangeably at the given times. Retrieving actions starts as soon as actions are guaranteed to be stored as shown in row *Stored Actions*. When this retrieval is completed on all nodes and the state is updated as shown in row *Stored State*, retrieving this updated state starts immediately. This gives the state retrieval operation a maximum time of $3 * l_{max}$ to complete. Since nodes permanently request the same area, they will have shortcut connections to the multicast root nodes of the area in the different zones after the first retrieve operation.

In total, the areacast probably needs two to three messages to reach all nodes in the area since area sizes in the destination zone should be similar to the area sizes in the source zone. The result of the retrieve will be returned in one hop using a shortcut. Therefore, $3 * l_{max}$ seems to be sensible timeout for area retrieve operations as three messages in a row should rarely take the maximum latency. If timing bounds get too tight and a lot of retrieve operations fail to obtain the necessary replica votes, l_{max} can always be adjusted accordingly. This also increases the cycle length of the update process.

As shown in figure 5.7, nodes are only guaranteed to store the state of time t_i between time $t_i + 3 * l_{max}$ and time $t_{i+1} + l_{max}$. When the first node completed retrieving all player actions, it will update its state to time t_{i+1} . The earliest theoretically possible time for this is right after the start of requesting actions $t_{i+1} + l_{max}$. After that, requesting an area might yield an inconsistent state containing objects that have already been updated and objects that have not been updated yet. Therefore, players trying to check whether they are seeing the correct state of the world can only do this during the state retrieval phase when the world state should be consistent.

Clock Skew and Drift Compensation

The clocks of the different nodes are only loosely synchronized within the bounds obtainable by NTP. When a node reads the current time to program a timer to start

the action or state retrieve at times t_i^a or t_i^s , the timer will fire around these times on the local clock. However, this time will slightly deviate from the true global time according to NTP accuracy. It is not a severe problem if the retrieval process starts a little late as still the other nodes should have completed their update and store operations. However, the retrieval process should not start too early to give other nodes the chance to complete their operations within the specified bounds.

This can easily be compensated by adding the maximum deviation t_d to the true global time to the programmed time. This way a node will start operations t_d later on average and $2 * t_d$ later in the worst case if it runs slower but was compensated for running faster. In this worst case, the node will itself have less time than the specified bounds to complete its operations. However, retrieve operations have a $2 * l_{max}$ and $3 * l_{max}$ bound which are less likely to be exceeded as multiple message need to be late. On the other hand, the actions store operation has got a l_{max} bound that could be violated more often as only one store message has to be late. Therefore, we opted to ensure retrieve operations never run too early.

If we also assumed a minimum latency for messages, this minimum latency might have been sufficient to compensate for operations starting too early. However, it would be harder to decide on a minimum latency as messages could be really fast on a local network. Actions are also stored on the local node meaning no communication is necessary for the local store. Consequently, we did not use a lower bound for message latency higher than the implicit zero bound.

5.3.6 State Initialization

The update process initially starts at time t_0 . At that time, no prior state can be retrieved. Thus, the world state at time t_0 is just defined statically in the game code. When the storage is running and t_0 is reached, the initial state is created out of nowhere. In the worst case, only one node is present having to create the state of the whole world. In practice, the operator should supply a sufficiently large number of initial nodes so the world state is distributed right from the start. Since all nodes will have the state at t_0 right from the start, they can start retrieving the state in the update area right after the start.

At t_0 , player characters could also be created automatically for every node. However, players usually want to customize their characters or add new player characters later on. Adding new characters must be an action leading to a new state including the created character like any other player action. This could be realized by placing a

special character for each player in the world able to spawn new player characters. He can use this special character to generate an action to spawn a new player character. This special character could also check whether the player still has got free character slots to limit his number of player characters.

5.4 Storage Maintenance

The storage has to ensure replicas of all data are updated correctly in all replication zones as nodes join and leave the network. Changes in the network change the responsibility areas of nodes and require replicas to be transferred between nodes while the update process is running. This section explains how the storage handles joining and leaving nodes.

5.4.1 Node Join

When a node a joins the overlay, its `JoinRequests` will be routed to its future neighbours and it will be integrated into the network exchanging `NetworkViews` with neighbouring nodes. During that time, the network is locally not stabilized and the nodes there will not be able to receive messages buffering them instead. Any queries to that area will not be answered but processing the queries will be delayed until the network becomes stable again. This reduces the chances of successfully completing a query before the requesting node hits its timeout depending on how much time the integration into the network takes.

Even when the network join is complete, the storage will not be able to answer queries correctly in the area of the joined node. After the network join, the responsibility areas of nodes have changed and messages will be routed according to the new network layout. The responsibility area of the joined node a is a Voronoi cell consisting of fragments that formerly belonged to the Voronoi cells of its neighbours, see figure 5.8.

The neighbours currently store data belonging to a while a is still missing data it is responsible for. At this time, a 's neighbours can still answer queries for areas overlapping their Voronoi cells while a cannot answer queries for its area successfully. It first needs to receive data it is responsible for from its neighbours. Therefore, after the network join, each of a 's neighbours in the same replication zone sends one *PushData* message with data now inside a 's Voronoi cell to a . Furthermore, a 's

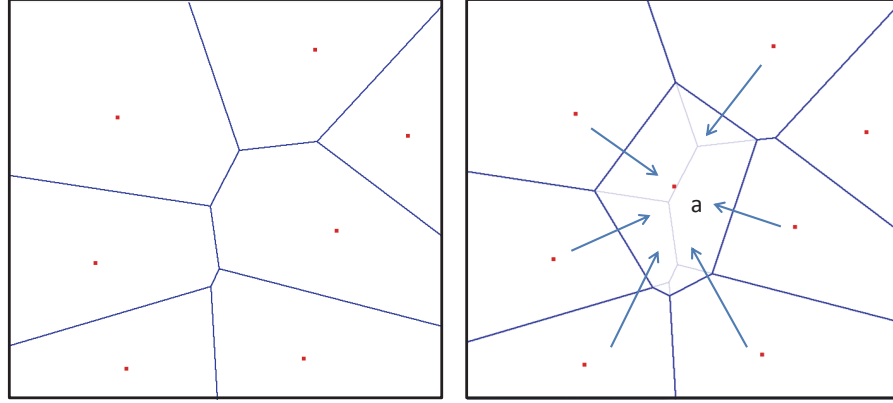


Figure 5.8: Node *a* receives PushData messages from its neighbour upon joining

neighbours will remove data they are no longer responsible for – data outside their maintenance areas.

Until *a* received a PushData message from every neighbour in the same zone, it will buffer all other messages coming in not processing any store and retrieve operations. However, the update process runs concurrently to *a*'s join. Node *a*'s neighbours update their world state at different times, depending on when they retrieved the last player's actions.

Pushing Data during the Update Process

If *a* joins during the action retrieval phase (see 5.7), it may end up with an inconsistent world state. Entities from one neighbour may already be updated to the new time while entities of other neighbours might still be old.

Even updating the world state from t_{i-1} to t_i synchronized among all nodes at $t_i + 3 * l_{max}$ would not be helpful. If *a* joins before the synchronization time, it will receive the old state of entities from time t_{i-1} with no way to update it to t_i at the synchronization time. To do this, it would need the state at t_{i-1} of its update area and all events of players in that area up to t_i .

Without synchronized updates, there is a chance for *a* all received PushDatas to be with state from time t_i if *a* joined sufficiently late in the action retrieval phase. Then, *a* can start retrieving the state of its update area for time t_i at time $t_i + 3 * l_{max}$. Furthermore, *a* will be able to correctly answer queries of other nodes requesting their update areas for time t_i . If *a* joins earlier in the action retrieval phase, it might receive an inconsistent state mixed of times t_{i-1} and t_i . Then, it cannot correctly

answer the queries of other nodes in the next state retrieval phase and the other replication zones have to compensate for this. Still, a can start retrieving its update area at $t_i + 3 * l_{max}$ and start its update process.

If a joins during the state retrieval phase of time t_i , it will take over a consistent state of time t_i and other nodes will be able to retrieve their update areas for that time. The join of a may only cause a delay in the answering of the retrieval in this replication zone. However, at $t_{i+1} + l_{max}$, when the action retrieval phase starts, a is missing its update area to decide on which actions to retrieve. Thus, it will not be able to calculate the state of time t_{i+1} at time $t_{i+1} + 3 * l_{max}$. In the next state retrieval phase, a will return only old state.

This means depending on when a new node joins, it may result in sub-areas of one replication zone to not be able to correctly answer retrieve operations for a certain time. The continuously running update process will fix this and requesting data from these sub-areas will work again correctly when the next update interval completed successfully. Until then, the other replication zones have to make up for this and provide the necessary reliability to answer queries. However, not all stored data is dynamically updated in the update process. There is also static data that has to be handled differently.

Static and Dynamic Data

The current world state consists of objects like player characters or non-player characters. These dynamic objects will be updated in the update process. On the other hands, the stored actions of players are static. Once they are stored, they will not be modified. They will just be removed when they are no longer needed to calculate a new world state. They will also not be refreshed as part of the update process.

Furthermore, at any point in time, the vast majority of players will not be in the virtual world. Their player characters will not be active and not perform any actions. Despite that, the storage has to store them reliably even if their state will not change as long as they are not in the virtual world. The state associated with a player character is much larger than the state of any other object in the world. In addition to the basic state like position and movement, player characters also have an inventory of items or state describing the progression of the character through various quests.

Since an inactive player character cannot interact with any other object in the world and given their data size, it would not make sense to include all player characters in

the state retrieval of the update process. Therefore, each player character has a flag marking it as active or inactive. If it is inactive, it will stay at its current position in the world but it will be invisible and no interactions with it are possible.

Inactive player characters and player actions are considered static data. A node executing a retrieve operation can decide to only include static or dynamic or both kinds of data. As part of the update process, only dynamic data will be retrieved. Static data will be retrieved when retrieving player actions. Static data will also be included in the PushData messages sent to a newly joined node.

Dynamic data is redundantly stored on neighbouring nodes if it is inside the overlapping parts of their maintenance areas. Using area retrieves, multiple duplicates of the same data can be retrieved decreasing chances of malicious nodes to tamper with data. This would not make sense for static player actions. They are retrieved using unicast messages that will only arrive at the one responsible node. Duplicates on neighbouring nodes will never be retrieved and would just waste space.

Therefore, a node a uses two kind of responsibility areas to decide on whether it is responsible for storing and maintaining a data item: its Voronoi cell A is used for static data and its maintenance area A_c is used for dynamic data. A node a pushing data to another node will always push dynamic and static data inside the other node's Voronoi cell and will afterwards remove dynamic data outside its maintenance area A_c and static data outside its Voronoi cell A .

The dynamic data a node received will be updated in the update process. Even if one of the neighbours was malicious and sent tampered data, this can be repaired if the majority of the other replication zones report a correct state on the next update. If a malicious node pushes tampered static data to a new node, there is no automatic mechanism to repair this. If at one point in time a malicious node happened to be responsible for a large area in one zone, all static data in that area it pushed to other nodes would always stay tampered. This is especially severe because it includes the state of player characters not in the world. Players want the storage to store their characters reliably while they are away. Therefore, we need a mechanism to deal with nodes tampering with static data in PushData messages.

Static Data Retrieval

The repair effect of the update process lies in periodically refreshing the state of the local replicas based on the replicas from all other replication zones. Therefore, to repair any tampered static data in PushData messages, a joined node also requests

the same data using a retrieve operation from the other replication zones. More precisely, a node a requests all static data from its Voronoi cell A after joining the overlay and adds it to its stored data.

To make sure player actions up to the time where a joined the network have successfully been stored, a delays this retrieve operation by l_{max} . This is the assumed maximum duration of the store operation. After joining the overlay, newer player actions will automatically be routed to a and do not have to be retrieved.

It would theoretically be possible to also retrieve dynamic data together with the static data. However, there would be no benefit in doing this as a really needs its update area that will be requested in the next update interval anyway.

Player Activation and Deactivation

When a player wants to leave the world, it can simply perform an action setting the state of its player character to inactive upon execution. During the update process, this action will be retrieved by the storage and all nodes storing a replica of a player character will set its state to inactive transforming it to static data.

To activate its player character, a player performs an action to activate the player character. However, nodes with replicas of that player character will not automatically fetch this action. They only fetch actions of player characters that have been active because inactive player character cannot perform any actions normally. The activation action is the only exception to this.

Therefore, the player sends a notification message to the nodes holding a replica of the static player character. This will cause these nodes to temporarily set the state of the character to active making it dynamic data. On the next state retrieval, the other node will fetch this temporarily activated character. This causes them to retrieve any actions this player character might have performed in the last interval. When they retrieve the activation action, they will consistently and permanently set the state of this player character to active until it will be deactivated again.

5.4.2 Node Departure

When a node leaves, its neighbours will take over shares of its responsibility area reversing the change shown in figure 5.8. The neighbours have to take over the data the node was responsible for. If a node performs a graceful leave, it would be

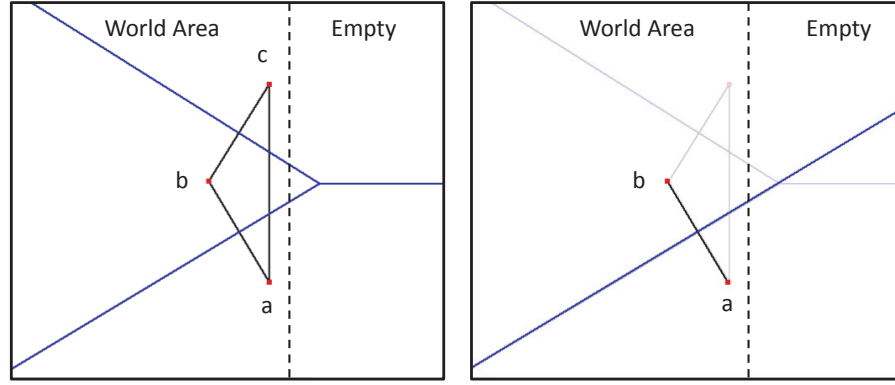


Figure 5.9: Failure of neighbour c does not add responsibility area to a

possible to prepare for the departure, pushing data to the neighbours and waiting for the right time to leave. However, nodes are generally considered to be unreliable and we expect them to leave unexpectedly just closing all connections. Therefore, we designed the storage maintenance procedure around this case.

The neighbours of the failed node will detect the failure and the self-stabilizing algorithm will repair the network structure. Once a neighbour gets a consistent network view with all its neighbours, it can calculate the additional area it is now responsible for and will start to obtain the data in that area. Similar to the join procedure, we treat static and dynamic data differently.

Retrieving Static Data

When a node fails, the replicas of static data in its replication zone are immediately lost as the Voronoi responsibility areas for static data do not overlap. To repair this, the neighbours of a failed node that have taken over parts of its responsibility area request this added area at the storage to retrieve it from the other replication zones. To make sure any store operations of player actions arrived at the responsible nodes, this retrieve operation is again delayed by l_{max} . Actions from after the fail time will already be routed to the correct neighbour that became responsible.

When all neighbours completed their retrieve operations, the static data in the area of the failed node will be available in its replication zone again. Until then, the other zones have to make up for the temporary unavailability of that data.

As shown in figure 5.9, not every neighbour of a failed node will get a share of its responsibility area. If a neighbour does not share a common border within the world area in the current zone, the added area is in the part of the current zone that does

not contain any data. Requesting this area will not yield any results and will be skipped. For neighbours within the world area, the added area will always be a convex polygon as shown in figure 5.8. Thus, it can be requested from the storage as retrieval of convex polygons is supported.

However, if multiple neighbours of a node fail simultaneously, the added area consists of multiple convex polygons that usually do not add up to a convex polygon. Instead of requesting multiple convex polygons, the node requests its own Voronoi cell completely reducing the number of retrieve operations at the cost of increased message size and more messages for this one operation. However, for reasonable churn rates this should be a rare case.

Retrieving Dynamic Data

When a node fails, only the dynamic data from the node's area not overlapping any maintenance area of neighbours is immediately lost. Depending on the size of the maintenance area in relation to the area of the failed node, it is actually possible that no dynamic data is lost in the node's replication zone. Therefore, we do not perform a dedicated retrieve operation for this data. Instead, any missing dynamic data will automatically be requested during the next update interval. However, if data was lost, nodes will not be able to retrieve that data from this replication zone and the other zones will have to compensate for it.

5.4.3 Effects of Churn

As shown, both joining and leaving of nodes may result in data becoming temporarily unavailable. The self-stabilization of the overlay and the waiting for PushData messages delays the processing of any retrieve operations. The inability to update its dynamic state after joining or the failure of nodes may lead to data replicas missing until the next update cycle is completed or the missing static data has been retrieved.

Redundancy by replication is the basic principle to deal with this unreliability. Static data will be made available again by retrieving it from other zones. Dynamic data will be retrieved from other zones automatically during the update process. This only works correctly if the worst case does not arise: the same data becomes unavailable in all replication zones. If there is a change in the network at nodes covering the same world area in all replication zones, data will be lost. The replication factor

must be sufficiently high assuming a certain maximum churn rate, so the probability of this happening is close to zero. We evaluated the reliability gained using this approach in section 6.4.

5.5 Resilience

Malicious nodes can tamper with any data they store and reply with this tampered data on any query they receive. Furthermore, they can also claim being responsible for data and answer with tampered data when they should have forwarded the query message closer to the node really responsible. They can never tamper with replies to responses of other nodes as they are always sent directly using shortcuts. The influence an attacker gains depends on the type of query performed as shown in the following sections.

5.5.1 Area Queries

When a malicious node receives an area query, it can claim to be the only responsible node if the message has not reached its target destination yet and the multicast for the message has not started yet. It can report any result about the current state of the world in the requested area. It can reply with any forged neighbourhood of existing node IDs to create any responsibility area. However, it should be careful not to create areas suspiciously big, like reporting only neighbours in other zones claiming responsibility for the whole zone. This might be used to detect and punish it. If it cannot claim being responsible, it could always drop the query message increasing its chances to tamper with the same query in other zones containing malicious nodes, too.

If the multicast has already started, it can only tamper with data in the area it is actually responsible for. It could also let the multicast fail by replying with a wrong neighbourhood or not replying at all. However, we assume it chooses to tamper if possible, as its primary interest lies in modifying the world state. A malicious node is only guaranteed to successfully tamper with data in one replication zone if that data is not inside the area overlapping with the maintenance areas of its honest neighbours. If it tampered with data in the overlapping part, conflicting versions will be sent back. If one version does not have a majority, the result from that zone will be discarded. So unless there is the rare case of two malicious nodes' maintenance area overlapping the maintenance area of an honest node creating the

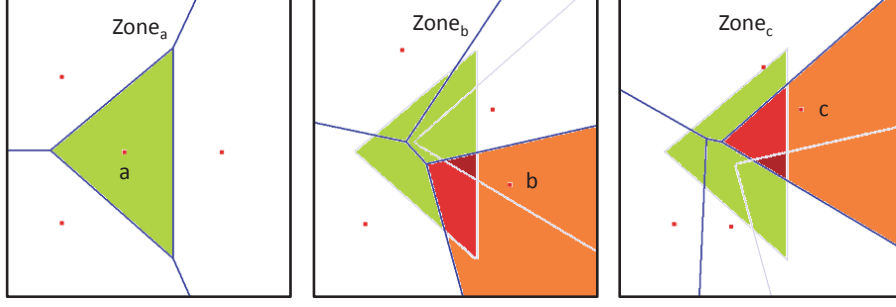


Figure 5.10: Node *a* requests its own area overlapping the areas of malicious nodes *b* and *c* in other zones

local majority, the malicious node can only hope to tamper successfully with data in the non-overlapping areas.

However, this is only true for dynamic data. The Voronoi responsibility areas for static data do not overlap and a malicious node can tamper with any inactive player character or actions of player characters it stored. This static data will be retrieved using an area query when a node joins, requesting the static data inside its Voronoi cell. It will also be retrieved by neighbours of a failed node retrieving their added responsibility area. To be able to tamper successfully, an attacker needs multiple malicious nodes in the majority of replication zones covering the same world area as shown in figure 5.10. There, node *a* requests its own area after joining and the requested area mapped to the other zones overlaps with the responsibility areas of malicious nodes *b* and *c* in the small dark red triangular area. If a malicious node just tampered with all data without caring about the results that will be delivered back from other replication zones, it risks being detected and punished. A clever attacker will try to prevent this and report tampered data only for areas where he can obtain a majority.

Obtaining a Tampered Majority

The scenario shown in figure 5.10 is not the only way to obtain that majority. In this scenario the attacker actually gained the authority over an area and will always be able to tamper. Due to the random distribution of nodes and with a sufficiently high replication factor, this is very unlikely to happen. However, he could also obtain a relative majority. If two malicious nodes in two zones are responsible for the same world area and the attacker controls malicious nodes in other zones to let the queries in these zones fail, it can still obtain a relative majority on the returned answers. It

would also be sufficient to tamper in one zone, intercept the query in another zone to create any forged answer, and let the queries in other zones fail.

The bigger the requested areas, the more nodes are involved in replying to a query. The more nodes involved, the higher the chances one of them is a malicious node able to let a query fail with the goal of obtaining a majority using the results from other zones. Therefore, not all types of area queries are equally susceptible to being tampered with.

The biggest area query is the state retrieve query in the update process, requesting a node's update area including everything that could have influenced its maintenance area since the last update. As we will see in section 6.4, these areas are much bigger than the other area queries. On the other hand, the overlapping of the maintenance provides additional resilience to tampering.

The second biggest area query is the Voronoi cell retrieval of static data when a node joins or when multiple neighbours of a node fail simultaneously. No overlapping with neighbours provides additional resilience for these queries.

The smallest area queries are the retrieve operations of the area of interest around the current position of a player character. These areas are usually much smaller than the above areas and thus least vulnerable to being tampered with. Even if they get tampered, the result is not severe because it means a single player will probably only temporarily see a tampered state. Permanent state manipulation by tampering the above state retrieval is much more severe.

Propagation of Tampered Data

If a node receives a tampered state for its update area during the update process, the updated state of its maintenance area will also be tampered. On the next update cycle, it will also report this state to other nodes giving the tampered state one more vote on all nodes requesting this area. However, this does not necessarily mean the node will always contain tampered state. It is possible the circumstances caused messages from zones containing the correct state to be late and their correct replies did not arrive in time. The tampered state was returned only because it obtained a relative majority due to the lateness of messages. Depending on the paths messages take and varying latencies of messages, the nodes in other zones might still receive a correct result for the same area. In the next update cycle, the node might be able to obtain the correct state again if the replies arrive in time.

However, it is also possible the one more vote the tampered state obtained is the decisive vote causing all nodes in other zones to retrieve a tampered state in the next update. Calculating their update even based on only one tampered object, the whole updated state can be considered as tampered because the one tampered entity can lead to all other objects getting an incorrect updated state. Then, nodes in all zones will contain tampered state in that area. Even worse, this tampered state will spread out through the world. In the next update, the neighbours of these nodes will request their update area and retrieve a tampered one. Calculating their update, their whole area will also be tampered and after a few update cycles, the whole world state will not be correct any more.

Tampered static data will not spread out in a similar way. A malicious node cannot modify or create actions of players as they are signed. It can only drop actions. Since actions lose relevance in the next update cycle and get deleted anyway, dropped actions will only spread to other nodes if nodes responsible for the same world area join or leave in the same update cycle. In this case, joined node *a* obtaining a result with missing actions may cause joining node *b* responsible for the same world area in another zone to not be able to retrieve all actions as well. When this happens in more zones, it can cause actions of players to get lost. On the next world update, these actions will be missing and the calculated state will not be correct. However, failure or join of nodes in multiple zones responsible for the same world area causes reliability problems anyway and is unlikely to happen.

If a joining node receives tampered inactive player characters, the replicas of these characters in that zone will be tampered. The node will report these tampered characters when other nodes join and leave and responsibility for the tampered characters in the other zones changes. However, this tampered character does not necessarily spread to other zones, depending on whether there are still enough zones with correct replicas and they can be requested successfully. Upon activation of the player character, it will become dynamic data and will be updated in all zones. Depending on how the first retrieval of the player character as dynamic data went, it might be tampered or correct. If most zones still contain a correct version and all zones are able to retrieve their update area correctly, the tampered player character will automatically be corrected.

Influence of Routing Paths

The number of nodes involved in a query not only increases with the size of the requested area but also with the length of the routing path. Both the retrieval of

dynamic data in the update process as well as the retrieval of static data on node join and leave target the same world area the node is responsible for. Due to the way the entries of the top routing table layer mirror the world position of a node in the other zones, requesting its own area will most likely lead to the first hop of the path already lying in the requested area starting the multicast. In addition to the nodes overlapping the requested area in the target zone, no other nodes will be involved. If the first retrieval was successful, any future retrieval will be routed using the established shortcut connections if the top layer routing table entry is not already inside the requested area.

In contrast to that, the area of interest requested by players to check the state of their local area is most likely far away from the world area the node is responsible for. The queries will be sent to the other zones on the first hop where they have to be forwarded with logarithmic path length until they reach their destination. Compared to requesting its own area, the number of involved nodes might actually be bigger increasing the vulnerability of this type of query. However, retrieving tampered results for the area of interest is not as severe as only one player is affected and the tampered data will not spread out.

5.5.2 Point Queries

Point retrieves are used to request actions of players. A node updating its update area retrieves the actions of all player characters in this area. Since player actions are signed, a malicious node can only try to drop actions of honest players. It cannot create new actions in the name of other players or tamper with existing actions.

Obtaining a Tampered Majority

After retrieving the actions of a player character, the requesting node will perform a majority voting on the result sets from the different zones. The goal is to decide whether an action was successfully triggered by a player. It is not sufficient to only show a player created the action using its signature. Then, only one node presenting the action would suffice. The majority of nodes responsible for storing this action have to agree whether the action arrived in time. Otherwise a malicious player can always wait for other players to take their actions before creating matching past actions giving him an advantage. Storing nodes will always discard late actions. If a majority of nodes does not include a certain action in their replies, this action is

deemed to not have been triggered. Therefore, a malicious node can try to obtain a majority not including certain actions of a player to undo these actions.

Since player characters are moving through the world, the set of player characters a node has to request when updating the world state will change all the time. It might have to request the actions of all player characters over time. Whether it can update the state in its area correctly depends on whether it can retrieve the actions of all player characters successfully. Being able to update correctly for some time might change when a new player character gets into the node's area it cannot correctly retrieve actions of. If multiple player characters are in the same area, not being able to successfully retrieve the actions of one player character leads to an incorrect update of the state.

Dropping the actions of one player character is probably not as severe as directly modifying the state of the world, because the effects on the world state of undoing some actions are limited compared to direct state modification. However, the updated state might actually diverge from the replicated states in other zones if the nodes there were able to retrieve all player actions correctly. On the next state retrieval, the votes for the correct state will be lower increasing chances for an attacker to obtain the majority for a state he tampered with in other zones. This tampered state can then propagate through the world.

Influence of Routing Paths

The first chance for a malicious node to drop actions appears when the honest player character stores its actions. A malicious node can drop any actions of all players it is responsible for. It could also not forward the store messages preventing the responsible nodes from storing the player actions correctly. However, the latter is unlikely to happen. Since a player stores its action at its own ID position and the top-level routing table entries mirror its position in the other zones, storing actions will most likely only take one hop.

The second chance to drop actions is preventing nodes from retrieving all actions of players in the area they are responsible for. A malicious node can simply drop the request message and all actions of a player during the last update interval will be dropped. If the malicious node is responsible, it can also select specific actions to drop.

Unfortunately, the actions an updating node retrieves depend on the player characters currently in the node's responsibility area. The points requested depend on the

player characters static ID position. This means they will be distributed all over the world area. Thus, the point queries will be routed with $O(\log(n))$ expected hops. The chances for malicious nodes to tamper with action retrieves are consequently higher than the chances to prevent the storing of actions. However, since there is no area overlapping multiple nodes to be requested at the end, it should still be lower than the chances of tampering with retrievals of the area of interest.

5.5.3 Preventing Eclipsing

After joining the overlay and filling its routing table a node a in zone Z_a retrieves the static data in its Voronoi cell A . It will send a retrieve message to each of the top layer routing table entries to request the area A mapped to all other zones. A top layer entry b receiving this message will most likely already be inside the requested world area A mapped to its zone Z_b . It will start the multicast and forward the request message to its neighbours.

From the requested Voronoi cell A , every receiver can actually calculate the neighbourhood of a . Furthermore, each node knows one node in every other zone because it stored it in the top routing table layer. Every receiver of a 's query in the Z_b can check whether its top layer entry for zone Z_a should be a neighbour of a . Due to the symmetric structure of top layer routing table entries, nodes close to b receiving the query because they overlap A mapped to zone Z_b are likely to have nodes close to a in their top-level entry for zone Z_a . These nodes are the correct neighbours for a . If a receiver notices a is missing a closer node as neighbour, it will notify a about this. Node a can exchange NetworkViews with this nodes and the self-stabilizing algorithm will cause a to join the overlay with the correct neighbourhood.

This scenario is also shown in figure 5.11. Node a joined the overlay in its zone but was eclipsed by the malicious node m_1, \dots, m_4 preventing him from obtaining its correct neighbours, the honest nodes h_1, h_3, h_4 . However, a filled its routing table successfully and b ended up as the top-layer routing table entry for zone Z_b . Thus, requesting its own Voronoi cell A , a sends a retrieve message to b . Node b overlaps this area and will start the multicast, forwarding this message to all its neighbours overlapping A mapped to Z_b . When neighbour c receives this message, it checks whether the neighbourhood of a indirectly reported through the area seems legit or if its top layer routing table entry for zone Z_a would be a better neighbour for the source of the message a . Node c 's top-layer entry being closest to c 's target position in zone Z_a is the node h_2 . Since h_2 is closer to a than its current neighbours, c

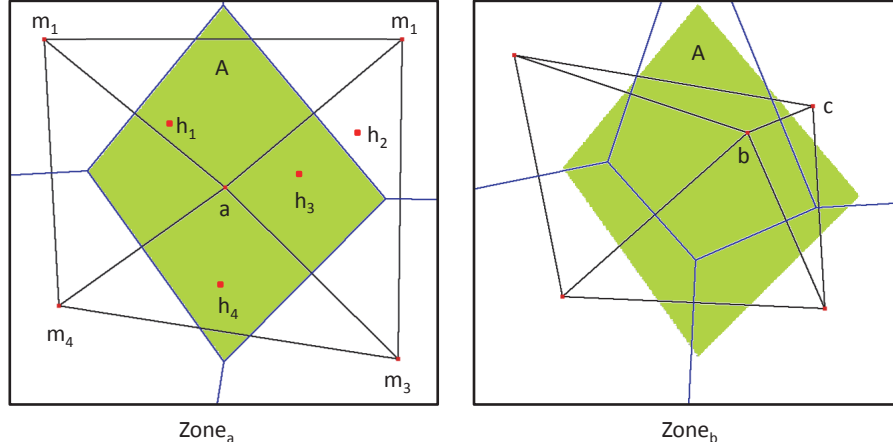


Figure 5.11: Node a is eclipsed initially but retrieves its true neighbourhood after querying its area A

notifies a about this. Node a contacts h_2 and begins to exchange NetworkViews. Thus, a will learn about the other honest nodes and will finally be integrated into the network successfully.

If a node a was eclipsed, it will retrieve a Voronoi cell bigger than the correct one with malicious neighbours farther away. More nodes in the other zones will receive this message increasing chances one of the receivers has a better neighbour for a in its top layer. Area A is retrieved in all zones and one zone alone reporting a closer node is enough to prevent a from staying eclipsed. As long as a has at least one honest node in its top routing table layer and this node has got honest neighbours with a correctly filled top routing table layer that will process the query, a cannot be eclipsed. Due to the highly redundant filling of routing tables, eclipsing is extremely unlikely to happen.

5.5.4 Preventing Shortcut Tampering

Obtaining a more than proportional influence on message routing is effectively prevented for the primary structure only allowing to connect to neighbours and for routing tables containing only nodes close to target positions. However, as shown in section 4.6 there is no such limitation in place for shortcuts. If a malicious node appears in the shortcut sets of all other nodes, it increases its chance to have messages routed its way. It will appear in these sets as a result of retrieving data at other nodes because they will return data using shortcuts that will stay open for a

certain time. This will be most effective using large area queries as multiple nodes will provide a part of the result on a shortcut connection.

Fortunately, the areas a node may request can be constrained. There are only two reasons for a node to request an area. The requested area can be related to its responsibility area which is requested in the update process or as part of storage maintenance. The requested area might also be the area of interest around the current position of a player character.

This information can be used by honest nodes to not answer queries that do not belong to the two categories. In the first category, the requested area is always around the node's own ID position. In the second category, the receiving node knows the state of the world in the area that is requested. It especially knows whether the requesting player character is really inside the requested area. Thus, a malicious player character would have to move through the world in order to create shortcuts on other nodes this way. Given the limited movement speed and the fact that shortcuts are only opened temporarily, this constrains the number of shortcuts a malicious node can keep open.

Limiting the areas a node can request can also partially prevent the information exposure cheat. Using this cheat, players try to circumvent the usually limited perception range of player characters by obtaining information about distant areas. This will only be possible for the responsibility area of a node, but not for any other area.

5.6 Discussion

The storage performs a replicated distributed simulation of a virtual world. Since fast-paced continuously updating the world state is impossible, updates are performed in regular intervals creating snapshots of the world state at certain times. These snapshots can be used by players to check whether the current state of the world they see is based on a correct state of the world as reported by the storage.

5.6.1 Reliability

To make the system reliable, the simulation is replicated in a configurable number r of replication zones. On each update, the nodes in each zone request the relevant information to calculate the updated state from all replication zones. This constantly

refreshes the stored state in all replication zones based on what is stored in all zones. In case errors occurred and the retrieved data is not consistent, a majority voting among the results from the different replication zones is used to return the correct state with a high probability.

Errors include store and retrieve operations taking more time than the assumed maximum time which is based on an assumed maximum message latency l_{max} . When these bounds are violated, some nodes might not be able to update the state correctly. This will be compensated for by the nodes in other replication zones as it is unlikely errors occur in the same world area in all replication zones.

Other reasons for consistency errors include nodes joining and leaving the system. When a node leaves, all of its data is lost. When a node joins, it might not be able to calculate the next updated state. This data loss will also be compensated by the data stored in the other replication zones. Data will be stored reliably as long as there is no data loss in the same world area in the majority of replication zones.

The replication factor r is the most influential parameter for the reliability of the storage. The higher r , the less likely the same world area is affected by errors in all replication zones. However, the assumed maximum message latency l_{max} also influences the reliability. Increasing l_{max} gives more time for operations to complete successfully in case of late messages. However, it also increases the update interval length. Thus, the system becomes more vulnerable to nodes joining and leaving and data being lost until the next update cycle refreshes the data in all replication zones.

5.6.2 Scalability

The scalability of the storage depends on the length of routing paths for messages and the number of messages transferred. As long as the number of messages is independent of the number of nodes and the number of nodes involved in transferring a message is in $O(\log(n))$ or a constant number, the system will be scalable.

Storing and Retrieving Player Actions

The overlay routes unicast messages in $O(\log(n))$ hops using routing tables in addition to the primary Delaunay structure. These messages are used to store actions

created by player characters and to retrieve the actions of all player characters in a node's update area.

A player character stores actions using its static ID position as key. Therefore, the replicas are always stored at the same nodes as long as there is no change in the network. After the first store operation, shortcuts will be established and store messages will be transferred in one hop. If r replication zones are used, r messages will be sent out.

When retrieving player actions, the retrieved actions depend on the player characters in the node's update area. On the first retrieval of each player character's actions, the path length will be $O(\log(n))$ and the result will be returned in one hop due to the established shortcut. As long as the player character remains in the area, future retrievals will also be routed in one hop.

The number of retrievals depends on the area size. The bigger the area, the more players are expected to be inside. With random distribution of node IDs, each node should have a $1/n$ th share of the key space area on average. This number has to be multiplied by r , as actions are requested from r replication zones. However, player characters are usually not distributed uniformly. There might be hotspots in the virtual world with a lot of players while other areas are empty. Nodes responsible for hotspots will have to request more player actions than nodes responsible for empty areas. However, players aggregating in hotspots cause scalability problems for all kinds of architectures, including server-based ones. If the world is designed to cause players to distribute sufficiently and not aggregate in hotspots too much, player character action retrieval should still scale.

Area of Interest Retrieval

The number of nodes involved in an area retrieve operation depends on the length of the path until the multicast is started and the number of nodes in the multicast tree. The latter depends on how many nodes' maintenance areas overlap the requested area. When a player requests its current area of interest, the request will take $O(\log(n))$ hops on the first request and one hop for any additional requests if he stays in the same area. The retrieved area is a fixed size circle – a compact shape with equal extends in the x- and the y-dimension. We argue the Voronoi cells of nodes also have a compact shape because node ID positions are picked by randomly filling the key space. The distances to the next neighbours in x- and y-dimension are more likely to be in the same range than distances in one dimension being

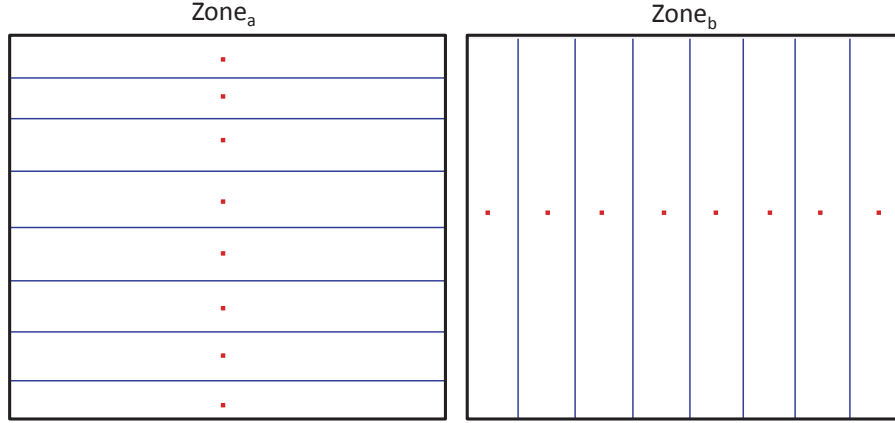


Figure 5.12: Nodes only distributed in one dimension lead to every Voronoi cell in one zone overlapping every Voronoi cell in the other zone

considerably larger leading to a stretched shape. Figure 5.12 shows a worst-case example of stretched shape, where every retrieval of an area in one zone would lead to the requested area overlapping the areas of all nodes in the other zone. However, usually Voronoi cells have the more compact shape as shown in figure 5.10 or figure 5.11 where Voronoi cells does not overlap all cells in another zone.

When both requested area and Voronoi cells have a compact shape, the number of overlapped cells can be estimated using the ratio of requested area size and average Voronoi cell size. The smaller the size of the Voronoi cells compared to the requested area size, the more cells will overlap. Since the Voronoi cell size decreases with increasing number of nodes in the network and the requested area of interest stays the same, the number of overlapping nodes actually increases with the network size.

However, the area of interest should usually be much smaller than the Voronoi cells of nodes for any reasonable network size. Thus, we mostly expect up to three nodes to be involved in the multicast for an area of interest retrieval if the retrieved circle overlaps the points of contact of three Voronoi cells. More Voronoi cells having the same point of contact is only possible if their centres are aligned on a circle which is unlikely to happen. Adding this constant amount of nodes involved in the multicast to the maximum $O(\log(n))$ nodes on the routing path yields a scalable $O(\log(n))$ number of nodes involved in an AOI retrieval. Again, the constant r has to be factored in as the retrieval is performed in all replication zones.

Update Area Retrieval

When a node retrieves its update area during the update process, the size of the requested area depends on the size of the Voronoi cell of the requesting node. The requested area is a the minimum enclosing circle of this cell with radius increased by the world-specific maximum influence range depending on effect propagation speed and update interval length. The size of the update area will be larger than the size of the Voronoi cell, but not by orders of magnitudes because the Voronoi cell has a compact shape, see also figure 5.5. Assuming this size factor to be constant, the size of the update area linearly depends on the size of Voronoi cells.

As we argued before, the number of Voronoi cells overlapping the requested area can be estimated using the ratio of requested area size to average Voronoi cell size for compact requested shapes. If update areas are just by a constant factor larger than Voronoi cells, this ratio will be constant. Increasing the number of nodes in the network, decreases Voronoi cell size and size of requested areas in the same way. No matter how many nodes in the network, the number of nodes overlapping a retrieved update area should remain the same. Since nodes keep requesting their own area, messages will only need one hop to reach that area using top-layer routing table entries. Therefore, the update retrieval of a nodes update area is a perfectly scalable operation, also taking the constant factor r for the different replication zones into account.

Consequently, requesting a node's own Voronoi cell with static data upon join or the added area in case of neighbour fail are also scalable as the requested areas are always smaller than the update areas.

Bandwidth Requirements

Whether node bandwidth is sufficient to complete operation within the assumed time bound l_{max} not only depends on scalability of operations with respect to the number of messages a node has to transfer. The number of operations in a given time frame and the message sizes also determine bandwidth requirements.

Message sizes requesting data or storing actions are very small as they do not contain much data. The only large messages are response messages containing world data. These will generally be transferred in one hop using shortcuts so they will only consume bandwidth of requesting and replying node. The amount of data transferred in one retrieval depends on the data describing the state of the world

in that area. With nodes retrieving their own update area and an assumed constant average amount of world state per area, the amount of transferred data on one retrieval actually decreases with growing network size and decreasing Voronoi cell size. However, at some point world size would have to be increased or more objects would have to be added to the world in order to accommodate larger amounts of players so this cannot be used to reduce traffic indefinitely.

The number of operations in a given time frame depends on different factors some of which we cannot influence. For example, the number of static data area retrievals depends on the churn rate we cannot control. The number of player actions stored depends on the activity of players and can only be influenced indirectly by modifying the rules of the world. However, the number of update area and player action retrievals linearly depends on the update interval frequency.

In every update interval, every nodes requests its update area and the actions of players in that area. If we halve the update frequency by doubling l_{max} , we would only transfer half of the messages, giving each message more time to arrive at its destination effectively halving the required bandwidth. However, this is not entirely true. Increasing interval length also increases the size of requested areas and the number of player characters contained therein. At some point, every node would have to retrieve the whole world area and the system would not scale. Furthermore, increasing the update interval also increases chances for the storage to lose data under churn. Since reliability is the primary concern, l_{max} should be kept low and high bandwidth requirements are just the price to pay to make the storage reliable.

Resilience

There is a large variety of ways malicious nodes can use to tamper with data. Malicious nodes can tamper with data they are responsible for and reply with tampered data when it is retrieved. Depending on the replication factor and due to overlapping maintenance areas, it is unlikely malicious nodes obtain an absolute majority for a sub-area of the virtual world.

However, malicious nodes can also obtain a relative majority by preventing correct results to be included in the voting. One malicious node that is involved in the query for a zone can always let the query fail by not answering the query or not forwarding a message it is supposed to forward. A correct result can only be obtained from a replication zone if no malicious nodes are involved.

In addition to that, malicious nodes can also prevent nodes that obtained a correct world state from calculating the correct updated state. Although they cannot create actions for other player characters, they can prevent nodes from retrieving all actions of a player character. When lying on the path of an action retrieve message they can just drop it. They can also drop the actions to be stored if these are routed their way. A node trying to calculate an updated state will miss player actions, calculate an incorrect update, and the malicious node has successfully reduced the number of votes for the correct state on the next update. In hotspot locations, just one of the many players' actions not being retrieved correctly leads to an incorrect state.

Replication of the world simulation in multiple zones is the main tool to counter the influence of malicious nodes. It should make it unlikely malicious nodes can successfully tamper in the majority of replication zones. Using shortcuts to transfer replies directly, malicious nodes can never tamper with the query replies of other honest nodes. If the first of multiple repeated queries ran correctly, shortcuts will be used for all following queries shortening paths and speeding up query completion increasing chances for correct replies to return before timing bounds are violated.

Considering the large number of ways malicious nodes can use to tamper, we expect the necessary replication factors to deal with a certain share of malicious nodes to be much higher than what is usually required to make a system reliable. This is especially true since replication also serves to compensate for the unreliability of individual nodes. It could happen malicious nodes tamper successfully only because some honest node's area failing at the right time in the right zone. Details on how many malicious nodes can be tolerated depending on the replication factor can be found in section 6.4.4.

Chapter 6

Evaluation

6.1 Network Simulation

To find out how well our system performs, we have to evaluate it with a number of nodes similar to current real-world MMVEs which means we have to test it with node number up to one thousand nodes. We can achieve test runs of this size only by means of simulation even if the final feasibility can only be shown in real-world tests [34]. Instead of running the system on real computers connected by a real network, we have to simulate the behaviour of nodes and the message transfer on the network.

There are well-known network simulators like ns-3 [102] or Omnet++ [117] with extension specifically targeted at simulating peer-to-peer overlays like OverSim [12]. Using these would have the big advantage of providing realistic network models while yielding results comparable to other works.

On the downside, these simulators require very specific ways the application under test has to be developed, e.g. by allowing only certain programming languages or even requiring specialized scripting languages nobody would use to develop real-world applications. Our goal was to implement our system in a way so it can be run in the real world, ideally by just exchanging real-world implementations of interfaces by simulator implementations. We wanted to prove our solution works in reality without hiding some real-world problems in a simulation-adapted implementation.

We implemented our system in C# because the C#-based XNA framework eases implementing real-time applications with 3D graphics. Furthermore, our implementation is based on the link layer implementation of a layered peer-to-peer architecture [65] also implemented in C#. Since there were no suitable network simulators

supporting C#, we had to develop a custom simulation. In the following, we will shortly describe the design of this simulator and its features to explain how we obtained our evaluation results.

6.1.1 Simulator Environment

The simulator defines interfaces for the application to interact with the environment. It provides communication facilities to connect to and disconnect from other nodes and to send and receive messages. It also allows a node to specify timed behaviour by programming timers and to obtain the current time so nodes can coordinate their actions based on the current time. The simulator also controls the lifecycle of nodes by invoking a start- and stop-function mimicking the user as it starts and exits the application. A bootstrapping procedure is also part of the simulated environment allowing a node to obtain the set of bootstrapping nodes to join the overlay and to register itself at the service to become a bootstrapping node for other nodes.

The actions of players and the movement of characters are not part of the simulated environment. Instead, the application is built to automatically generate player actions and movements according to some specified scenario.

The provided interface only provides asynchronous non-blocking functions. An application may not wait for the establishment of a connection or the reception of a message. After invoking a function to establish a connection, the function returns immediately. Connection establishment is signalled via events or callback functions. Internally, the simulator only schedules events to decide when a callback has to be executed on the different nodes.

This way the system can be simulated in a single thread. Therefore, operating system scheduling of threads does not interfere with the order of operations executed by the scheduler. The simulation becomes completely deterministic. Simulation runs can be repeated and will deliver the same results with the same settings. Pseudo random number generators with seeds as part of the simulation settings serve to generate deterministic random behaviour. This determinism is also tremendously helpful when developing a large decentralized system to understand the system and to find implementation errors.

When the simulation is started, the simulator will use the provided configuration parameters to initialize the simulation setting. It initializes properties like random distributions and random seeds, instantiates the configured number of nodes with

their application code, and schedules initial events. Then it starts processing these events.

6.1.2 Event Processing

Internally, the simulator performs a discrete event-based simulation [45] similar to the one used to realize consistent simulations of a virtual world. It uses a single global event queue containing all future events to be processed. Each event has got an execution wall time and events in the queue are ordered in time-ascending order. Every event is associated with a node. When executed, the event happens at that node. An event has a type specifying what operation will be performed when the event is executed.

The simulator starts its main loop by searching for the first event in the queue. This will be an event of type *StartEvent* that has previously been added to the event queue as part of the simulation setup. The simulator will advance its current global time to the time of the event and will execute it. A *StartEvent* will be executed by invoking the *start()* function on the corresponding node. Upon invocation of that function, the node will perform any actions necessary to initialize itself and to start running.

The node will start joining the overlay by invoking a function of the bootstrapping service to obtain bootstrapping nodes. The node also passes a callback function to be invoked when bootstrapping is completed. The bootstrapper function is implemented by a component of the simulator. Thus, the invocation will pass control to the simulator. The simulator will then create an event of type *BootstrappingCompleted* initializing it with the node and its callback function. Then, it schedules this event assigning an execution time and adding it to the event queue. The difference between execution time and current global time is the duration of the bootstrapping process of that node.

After adding the event, the bootstrapping function of the simulator returns to the start-function of the node. This function could invoke more functions of the simulator, each of which will add more future events to the event queue. Our application's start function will simply return to the simulator's main loop. The node will continue running when the simulator finally invokes the callback function in the *BootstrappingCompleted* event. The simulator's main loop will then search for the next event in the queue and process it by invoking a function on the corresponding node, which will in turn invoke functions of the simulator to generate and schedule events.

This *fetch(event)–execute(event)–callback(event.node)–schedule(events)* loop is the basic cycle performed by event-based simulators. Combined with wall time, it allows to measure the durations of all environment operations implemented by the simulator. The durations depend on the scheduling of events. However, the durations of calculations on individual nodes is not taken into account. Time in the simulation only passes when the simulator advances to the next event in the queue. When a node takes a long time to perform an expensive calculation, this calculation still happened instantaneously in one step from the perspective of the simulator – although the simulation will take that additional time. However, in reality the processing of messages should not be computationally expensive compared to the rendering of a 3D-world and there should be enough computational power to process any callbacks of the simulator reasonably fast.

In addition to the non-existing limit on the amount of calculation a node performs in a callback, the simulator also does not enforce a limit on the amount of environment functions a node can invoke in a callback. A node could send out thousands of messages in one step with memory for the event queue being the only limit. In practice, operating system message buffers will probably provide a lower limit. However, message transfer events will be scheduled in a way that all messages will be transferred sequentially taking the available bandwidth into account.

This means simulated applications have to be well-behaved in the sense of not doing unrealistic amounts of work in one step (one callback function) to provide realistic simulation results. Furthermore, they may only use the functions provided by the simulator to interact with their environment.

Despite being single-threaded, the simulator simulates parallel execution of operations on different nodes. The trick is while parallel operations are running – e.g. messages are being transferred – the simulator does not have to perform any calculations. It just schedules an event when the operation is started and executes the event when the operation has finished. The sequential processing of events in the queue creates a serialization of all events in the system. Thus, events totally ordered. This is just one of multiple possible interleavings of parallel actions on the different nodes. Different scheduling decisions, e.g. because of different random seeds, would create different serializations of events.

In the following sections, we will present the models underlying the scheduling decisions of the different operations like network communication and timed operations.

6.1.3 Network Communication

The main environment communication functions are connection establishment, disconnection, and message sending and reception. When a node wants to send a message to another connected node, the simulator has to schedule a reception event on the receiving node. The time between sending and receiving is the latency of the message.

Message Transfer

There is a wide range of different network models to predict the latency of messages passing through the Internet. Some apply queuing theory modelling the transfer of packets through the Internet as a series of queues passed by the packet at the different routers [84]. Empirical approaches perform latency measurements embedding nodes into a higher-dimensional vector space using the Euclidean distance function to predict message latencies [33]. Other approaches fit random distributions like the Weibull [61] or the Pareto [131] distribution to measured network latencies.

We decided to use a simple hybrid model combining a random distribution to model Internet latencies with FIFO queues at sender and receiver to model the bandwidth-limited up- and downstream of DSL-connected hosts. The first part provides varying message latencies and out of order delivery modelling congestion and routing via different paths. However, the random distribution models message latency independent of the message size. Since we expect the bandwidth of ADSL-connected PCs, especially the upstream, to be a bottleneck handling the high amount of traffic caused by replication, we added FIFO queues modelling the upstream and downstream channel of these hosts.

When a node sends a message, the message first passes through the upstream channel. If the channel is currently in use by another message, the message stays in the queue until the channel is free. A message blocks the channel for a time equalling the message size divided by the bandwidth of the channel. When the message has passed through the channel, an Internet latency is drawn from the random distribution to calculate the time it arrives at the downstream channel of the receiver. There, messages are again queued when the channel is occupied by other messages. When the message finally arrives at the node, it is first checked whether any older messages are still in transit. If not, it is passed to the application. Otherwise, it is buffered until the older messages arrived and are passed up. Consequently, FIFO

message order is preserved. This is consistent with our system model and our real implementation of the link layer using TCP guaranteeing FIFO order.

We implemented this model by scheduling four different types of events within the simulator. When a node invokes the function to send a message, the simulator schedules a *MessageSend* event with a small constant delay to model the internal computation that happens when sending a message. When processing this event, the simulator schedules a *MessageSent* event modelling the passing of the message through the upstream channel. Therefore, this event is scheduled for execution at the next time the upstream channel is free plus the time to transfer the message using message size and channel bandwidth. The channel is marked as being occupied accordingly. When the *MessageSent* event is processed, a *MessageTraversed* event is scheduled drawing a latency from the Internet latency distribution. Processing this event creates a *MessageReceived* event which is scheduled the same way as the *MessageSent* but using the downstream channel of the receiving node. Processing the *MessageReceived* event passes the message to the reorder buffer which in turn passes the messages up to the application in FIFO order of message sending.

Based on our network model, we can record all messages, their size, and the message latency. However, our model is not a precise simulation of TCP. It only captures the major characteristics of real TCP/IP link layer implementations: limited bandwidth and varying Internet message latencies. The biggest deviation from reality is the missing splitting of messages to comply with maximum transfer unit (MTU) sizes. Every message is handled as one message with regard to its Internet latency independent of its size. The only influence of message size is up-/downstream channel transfer time. We also do not model packet loss, TCP acknowledgements, message retransmits, and the traffic caused by these additional messages. The increased latency due to message loss is subsumed in the Internet latency random distribution. Including this handling with more precision would mean reimplementing a whole TCP/IP stack with all its timers and flow control algorithms which would be infeasible.

When deciding on the latency distribution to use, we settled with the Pareto distribution. There are a lot of controversial results about which random distributions capture packet latency best and which parameters to use. The Pareto distribution is used in some works e.g. [131] and it stood out because of its simplicity and it is computational inexpensiveness. Furthermore, its two parameters k and x_{min} define a lower latency bound and the expected value of the distribution. The cumulative distribution function of the Pareto distribution is defined as $F_{Par}(x) = 1 - (\frac{x_{min}}{x})^k$

for $x \geq x_{min}$ and $F_{Par}(x) = 0$ for $x < x_{min}$. Therefore, x_{min} is the lower bound for latencies. The expected value of a Pareto distribution is defined as $E(Par) = \frac{k*x_{min}}{k-1}$ for $k > 1$ and $E(Par) = \infty$ for $k \leq 1$. Thus, the parameters of the Pareto distribution can be calculated from a desired lower latency bound and expected value of latencies.

The Pareto distribution is a heavy-tail distribution. This can be informally characterized as a distribution with a larger part of the weight (the area under the probability density function) in the tail of the distribution which is the area of high latencies and low probabilities. This means although the probability is very low, it will generate latencies as big as hours, days, weeks, and years. In the Internet, a packet will never travel this long as it will be dropped when its TTL reaches zero. Furthermore, it will be retransmitted when the retransmit timeout kicks in. Therefore, we use a truncated Pareto distribution also specifying a maximum latency. This kind of distribution is commonly used when there is a natural upper bound on the probability tail [1].

In [131], the Pareto distribution models packet latency. As mentioned before, we use it to model latency of arbitrarily sized messages. In reality, packet latency is not message latency. Therefore, our results will only match the results of [131] suggesting similarity to real-world latency for small messages only. Our results will differ for large messages. However, it was most important for us to model varying message latency and bandwidth limitation and we did not focus on most precise capturing of real-world latency.

Message Layout

A node can send a message to another node with a certain link layer address. The message payload is a byte sequence. The size used by the simulator to calculate the message transfer time consists of the number of payload bytes plus header information. In the link layer model [65] we used, the link layer allows to establish logical connections between arbitrary peers. As peers might be connected to the Internet using NAT routers, relaying is the only possible connection strategy in the worst case. Therefore the link layer features its own header containing internal and external TCP/IP endpoints of sender and receiver and additional information to establish a logical connection. This header information sums up to 74 bytes. Furthermore, we also consider the TCP/IP header as part of the message size adding another 40 bytes to the message size. For small payloads like RtRequests, this might actually create a considerable overhead.

Connection Handling

Before nodes can exchange messages, they have to establish a connection. The connection establishment is simulated using a three-way handshake of TCP/IP control messages followed by a three-way handshake to exchange certificate information and agree on a common symmetric key for data encryption as implemented in the link layer. Therefore, establishing a connection might take a considerable amount of time in the simulation, depending on the individual message latencies. However, we did not include procedures like hole-punching [66] and relaying that would cause additional delay.

Disconnection is also performed using a three-way handshake like TCP/IP. If a node with open connections fails, there is no automatic notification of the other nodes. Instead, the next message sent to the failed node causes the simulator to send a TCP reset message back to the source when the message arrives at the failed node. In practice, this is not guaranteed to happen and node fail could be detected on timeouts only. However, we used this simpler version to not have to implement retransmit timers of the TCP stack. Since node fail is only detected on message sending, it might actually take some time for a node to detect the failure of a connected node. Therefore, the simulator allows nodes to program a ping interval for each connection and the simulated link layer will automatically generate ping messages and answer incoming pings with pong messages. Round trip times will be calculated automatically and can be read by applications similar to the real link manager implementation. Furthermore, connected nodes can detect node failure on ping messages if no user traffic happens in that time.

Since connection and disconnection takes some time due to the message exchange, certain special cases can arise. Nodes might try to connect to each other in parallel or disconnect in parallel. A node might try to connect to a node while a disconnection is in progress and vice versa. This happens because nodes do not necessarily have the same knowledge at the same time and might change their decisions on whether they need a connection at any time. These cases are also handled correctly by matching parallel handshakes to each other and delaying new handshakes until current handshakes have been finished.

6.1.4 Time and Clocks

Nodes can read the current time using a clock provided by the simulated environment. Furthermore, they can perform actions at certain times by registering a timer

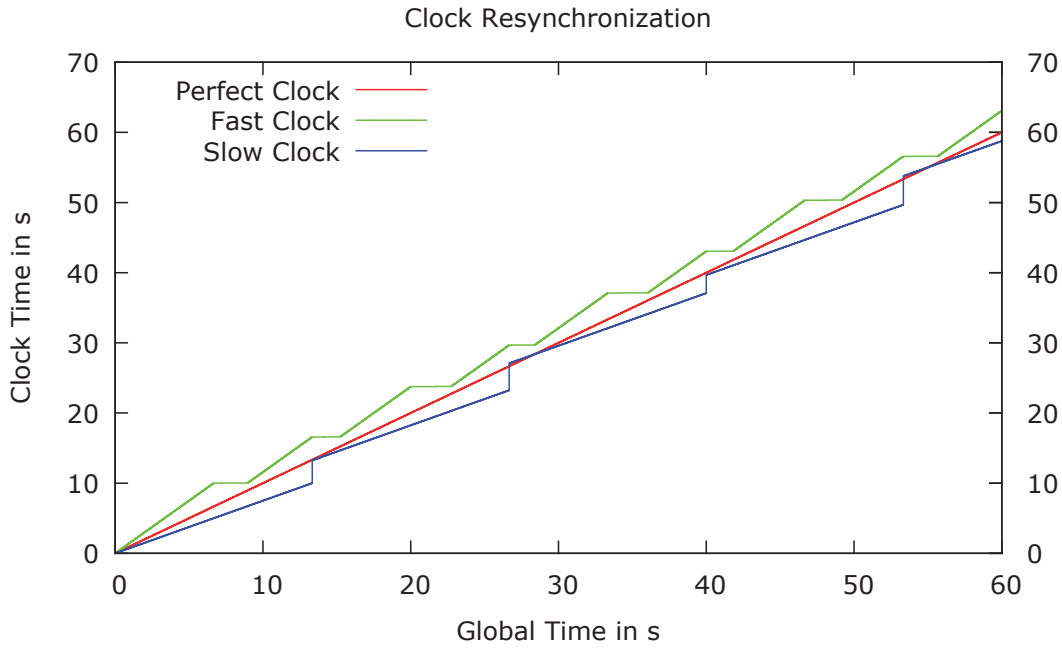


Figure 6.1: NTP-inspired clock resynchronization of drifting clocks

callback function at the simulated environment to be invoked after a specified time has passed. Using clocks and timers, nodes are able to coordinate their actions.

However, in reality clocks are never synchronized and we need to show our system also works when clocks are not perfectly synchronized. In our system model, we assume clocks being synchronized within bounds that can be realized using NTP as clock synchronization algorithm. Therefore, we have implemented the simulated clocks to drift with a constant drift rate away from the global simulation time. The drift rate is different for each node and randomly picked from a given bounded interval. Furthermore, each node resynchronizes itself with the global system in fixed intervals. We assume NTP is able to synchronize within certain bounds and the obtained time is normally distributed around the global time. If the obtained time is ahead of the time on the local clock, the clock is directly adjusted to the obtained value. If the obtained time is behind the local clock, the clock is not set back but is runs with a drastically reduced speed for the time difference to the obtained time. This way the local clock always runs within an interval of the global time. Figure 6.1 shows an exaggerated example of a fast clock running at 1.5 times normal speed and a slow clock running at 0.75 the normal speed and a resynchronization accuracy of one second so clock differences become visible.

The resynchronization is scheduled every 10s. However, resynchronization is scheduled using the local timer which is drifting according to the local clock drift. Therefore, the fast clock performs resynchronization more often than the slow clock. Furthermore, the phase of reduced clock speed is not able to actually catch up with the global time as its lengths is also determined using a local timer.

Local timers are not adapted when clock values are synchronized, consistent with the timer behaviour of operating systems. If a programmer specifies a timer to run in one second, he would rather accept a small deviation from the one second because of clock drift than the timer running in two seconds because the clock was synchronized and stopped for a second. As long as timers are programmed with short time spans, drift will not have a major impact.

If timers are used to coordinated actions of nodes run at a certain clock time, it is often desirable to have all nodes start their actions no earlier than the given time. This can be achieved by incorporating the maximum clock drift and the minimum synchronization accuracy in the programmed time span. To mitigate the larger deviations created when the target time is still far away, two timers can be combined. The first is set in a way that it will run shortly before the target time and the second timer is programmed with the shorter remaining time span.

In our simulation, we used a drift bound of maximum 1% speed deviation from the perfect clock, a resynchronization accuracy of 10 ms, and 10 s resynchronization interval yielding a much tighter clock synchronization than shown in figure 6.1.

Internally, reading the clock just maps the current global simulation time to the local time of the node incorporating clock drift and normally distributed synchronization accuracy to get NTP-like behaviour. NTP is not simulated directly so no messages are generated for clock synchronization. Timers are implemented by scheduling a *TimerEvent* in the global event queue for a future time considering the drift of the local clock. When this event is executed, the callback functions will be invoked on the respective node. It is also possible to use periodic timers by scheduling *PeriodicTimerEvents* automatically scheduling new events on execution.

6.1.5 Node Lifecycle

During simulator initialization, a configurable number of nodes are instantiated and a *StartEvent* is scheduled for every node. Furthermore, it is already decided when the node will leave the network by scheduling a *StopEvent* in case the user exited the application. A *FailEvent* is scheduled to denote a silent fail e.g. because of a

system crash at the user or a disconnection from the Internet. The execution times of these events can be preconfigured to generate specific test scenarios. However, it is more common to generate scenarios by specifying a churn rate and use random distributions to decide on the node start and stop time.

Churn can be modelled in terms of two probability distributions [62]. The first is the inter-arrival distribution (IAD). It specifies how much time passes between the arrivals of nodes in the network. The second is the session length distribution (SLD) specifying the duration a node stays in the network. Experimental results in peer-to-peer file sharing networks suggest the exponential or the Weibull distribution can be used to describe churn [114].

Internally, executing the `StartEvent` invokes the callback function on the node so it can initialize itself and start running. Executing a `StopEvent` invokes a stop function on the node giving it the chance to shut itself down. Finally, the node can invoke a stop function on its simulated link layer closing all connections to other nodes and preventing the node from receiving any further messages. By closing all connections, other nodes notice the departure of the node.

The `FailEvent` on the other hand simply removes all scheduled events at the failed node. Any events of received message events or timer events will be removed so the node will never run again. Messages towards that node will be answered by connection reset messages signalling the failure of the node. Thus, other nodes will detect the failure when next sending a message to that node, e.g. a ping message.

6.1.6 Bootstrapping

The simulator features a simple simulated bootstrapping service. In its start function, a node usually invokes a function of the simulated bootstrapper to receive its bootstrapping peers passing a callback. When this happens, the simulator schedules a `BootstrappingCompleted` event for a random time taken from a configured interval. Upon execution of this event, the simulator randomly picks nodes from the overlay and returns them to the node invoking its callback function.

The bootstrapping service is actually aware of zones in the overlay and is able to supply one node from each zone at random. The bootstrapping service is not perfect in the sense of guaranteeing to return at maximum a number of malicious nodes proportional to the share in the whole network. However, due to the random selection it is unlikely it returns more than this number of nodes. Therefore, it is ideal with respect to what could practically be achieved by a bootstrapping service

knowing all nodes in the system. In practice, if a bootstrapping service maintains only a subset of all nodes, it might be more vulnerable to malicious nodes obtaining a bigger influence.

When a node joined the overlay, it registers itself at the bootstrapping service to become a potential bootstrapping node for others. If it is stopped, it also de-registers itself. If it fails, the simulator also takes it out of the set of potential bootstrapping nodes.

6.2 Common Simulation Parameters

The simulation features a large set of configuration parameters to initialize the simulator and the test scenarios. It is not possible to systematically evaluate the effects of all parameters. Some of them had to be picked based on assumptions and common sense. These parameters were the same for all simulation runs. However, we believe we picked parameters good enough to obtain realistic results and picking different parameters would not deliver qualitatively different results. Usually, we used more conservative assumptions than probably necessary.

6.2.1 Communication Parameters

As mentioned before, the Internet message latency is calculated using a Pareto distribution with a minimum latency of 20 ms. Every message transfer takes at least that time plus the time to pass through up- and downstream channels. The distribution parameter k is picked as 1.5 yielding an average latency of 60 ms. We truncated the Pareto distribution at a maximum latency of 5 seconds. In an Internet settings, most message will be much faster and TCP retransmit timers can always be adapted to retransmit sooner to get lost messages through in that maximum time with a high probability.

At first, the bandwidths of nodes were set using a scenario where most nodes had an ADSL connection with upstream bandwidths down to 192 KBit/s. We quickly recognized this to be a major bottleneck with messages in the upstream channel being delayed by several seconds. Therefore, we decided not to systematically evaluate the influence of this parameter because for most settings this would just mean the storage could not work correctly. Instead, we picked a fixed symmetric bandwidth

of 100 MBit/s that should be realistic in a few years as fibre Internet access becomes more common.

The link layer added 42 bytes of TCP/IP header data and 74 bytes of link layer data to each transferred message. During the certificate exchange on connection establishment, the three exchanged messages had a size of 1028 bytes, 1060 bytes, and 32 bytes respectively in line with the used real link layer implementation.

6.2.2 Timing parameters

Errors in physical processes like measuring time are typically modelled using normal distributions in absence of any additional information. Therefore, the clock drift of each node is drawn from a normal distribution with mean value one – the speed of the perfect clock. The standard deviation is set at $\sigma = 0.1\%$. Therefore, clock drift is practically bounded at $3 * \sigma = 0.3\%$ using the common estimation that 99.73% of all values lie within $3 * \sigma$. The maximum allowed drift rate is 1%. This is a very conservative assumption as nowadays the timestamp counter bound to quartz crystal oscillations is used as a timing source yielding much lower drift rates [88].

The resynchronization difference to the perfect clock of each node is calculated by drawing a value from a normal distribution with mean value zero on each resynchronization. The standard deviation is set to $\sigma = 3ms$ so clocks can be synchronized within a maximum distance of $3 * \sigma = 9ms$ to the global time. Resynchronization is scheduled to happen every ten seconds, which is the minimum interval time allowed by the rules of NTP.

6.2.3 Lifecycle Settings

Regarding the arrival and departure of different nodes as independent, we decided to use the exponential distribution for both the inter-arrival distribution (IAD) as well as the session length distribution (SLD). We fixed the expected value of the SLD at two hours regarding this as the average time a player stays in the system. A study of MMORPG session lengths [97] suggests this is a reasonable value. We varied the IAD expected value creating an equilibrium of node join and departure at different node numbers in the network. Figure 6.2 shows the number of nodes in the network over eight hours with an expected value of 25 seconds for the IAD. Ten different random seeds were used to create ten scenarios with an equilibrium between 250 and 300 nodes after eight hours.

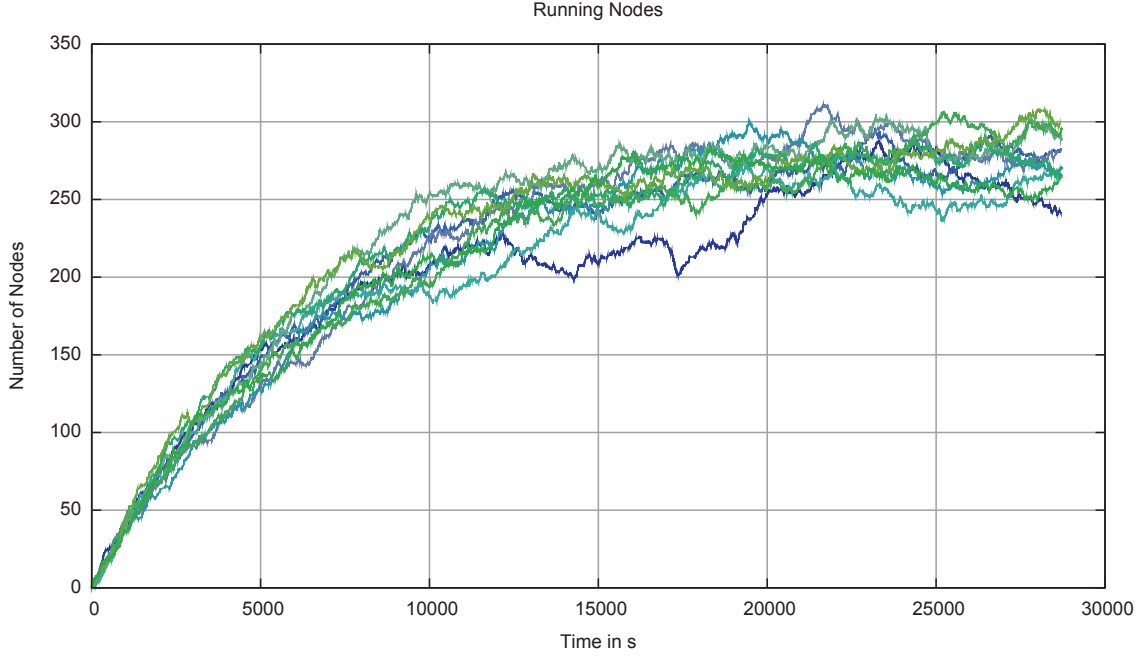


Figure 6.2: Number of nodes in the network with $E(SLD) = 2h$ and $E(IAD) = 25s$

Node departure was realized as graceful stopping of nodes instead of node failure. The ping messages to detect node failure within reasonable time quickly dominated the traffic and slowed down the simulation as 1000 nodes exchanged messages on each link several times a minute. In reality, people usually at least shut down their computer with the operating system closing all open connection rather than just turning it off. Furthermore, varying latency of close messages still means connected nodes do not immediately notice the departure at the same time. Therefore, this scenario is still reasonably realistic.

6.2.4 Bootstrapper Settings

The time the bootstrapping service needs to return bootstrapping peers is taken from a uniform distribution on the interval from 100 ms to 300 ms. The time for the bootstrapping service to add a node to the set of potential bootstrapping peers and to remove it when it leaves is also drawn uniformly from the interval 200 ms to 500 ms. The delays are mainly introduced to not rely on the bootstrapping service to always know the global state of the network. The delays may cause scenarios where a node has already left but is reported as a bootstrapping peer to new nodes. Furthermore, the bootstrapping service might exclude nodes from the selection that have already started to register themselves as potential bootstrapping peers.

6.2.5 Overlay Settings

In all test scenarios, 5000 node IDs were randomly generated first. This was the maximum number of nodes that could join the network in one test run. However, only a subset of these nodes was in the network at the same time in all scenarios. The virtual world size was $6700m \times 6700m$ yielding an area of $44.89km^2$. This is roughly equal to the size of one continent in World of Warcraft.

However, picking IDs at random does not create even distributions where all nodes have similar distances to each other and Voronoi cells have similar sizes. Instead, nodes form clusters in some areas giving them very small Voronoi cells whereas only a few nodes with large cells are located in other areas. Therefore, we put a rasterization of 5000 IDs on the space guaranteeing a minimum distance between nodes. This way we can also ensure no two IDs have the same integral part. However, since only a fraction of all nodes are in the network at any given time, this random sub-selection still creates variations in the sizes of Voronoi cells.

No matter which replication factors were chosen in the different scenarios, the number of routing table layers was always set to four – despite IDs having different lengths. Considering how sparsely populated the ID space is, additional layers would have stayed empty for the most part and would only have caused unnecessary traffic.

Finally, shortcut connections are kept open for a maximum of 90 seconds without traffic in all test scenarios.

6.3 Overlay Evaluation

This section presents our evaluation result comparing these results with our initial requirements. We will present results on the structure of the created overlays over multiple test scenarios suggesting it can serve as a reliable base for the storage. Afterwards, we show the overlay features a scalable maintenance mechanism. Finally, its resilience against attackers is shown by adding various numbers of malicious nodes to the test scenarios.

6.3.1 Overlay Test Scenarios

In the overlay test scenarios, the storage runs on top of the overlay without any stored data and without performing any queries. The only exception is performing the initial query for the nodes responsibility area as this is necessary to make sure a node is not eclipsed. We tested the network maintenance under several churn scenarios. We used 14 different scenarios with varying churn rates reaching from $E(IAD) = 6s$ to $E(IAD) = 25s$, to evaluate reliability and scalability. In [97] current population on a World of Warcraft server was shown to vary between 400 and 1600 players and using these values for the IAD created similar populations in our test runs.

Each scenario was tested with the three different replication factors $r = 4; 9; 16$ with $(r_x = 2, r_y = 2)$, $(r_x = 3, r_y = 3)$, and $(r_x = 4, r_y = 4)$. We picked these to have one set of tests with only light, medium, and heavy amounts of redundancy. We used identical values for r_x and r_y so the ID space structure and thus the layout of the routing tables is identical with different replication factors.

One test run simulated eight hours of overlay maintenance. Each scenario was run ten times each with different random number seeds. Thus, all random behaviour like start and stop times and message latencies were different on the runs of one scenario. However, for the same run of a scenario with different replication factors, the times at which nodes join and leave are exactly the same. The nodes will be at different positions as the ID space is different for different replication factors. The amount of running nodes at any given time is identical making it easier to compare the results for the same run under different replication factors.

To test resilience, we later added varying numbers of malicious nodes to one of the scenarios. In total we ran 420 tests to evaluate the influence of churn and additional 240 tests to evaluate the resilience. One test run lasted between 0.5 hours and 6.5 hours using one core of our Intel Xeon X5650 server running at 2.67 GHz. This limited the number of tests we could perform in a reasonable amount of time.

Table 6.1 shows the number of nodes that have been started over eight hours for each test run. Table 6.2 shows how many nodes have been in the network when eight hours were over. This number can be used to estimate the churn equilibrium – the average number of nodes in the network as it continues running. Even for identical settings, there is still some variation in the number of started and running nodes in the different runs. This can also be seen in figure 6.3 showing the number of nodes in the network for the IAD6 scenario similar to figure 6.2 showing this for the IAD25

Table 6.1: Number of started nodes in each run

IAD	\bar{n}	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
6	4799	4787	4771	4716	4767	4750	4769	4929	4846	4864	4789
7	4129	4106	4097	4095	4070	4099	4253	4190	4158	4151	4073
8	3610	3594	3597	3596	3545	3588	3713	3658	3633	3637	3542
9	3195	3185	3183	3205	3155	3169	3180	3292	3233	3211	3134
10	2874	2823	2851	2860	2836	2855	2972	2914	2924	2877	2827
11	2616	2567	2595	2595	2597	2595	2700	2656	2676	2639	2540
12	2407	2359	2393	2412	2368	2383	2489	2422	2491	2436	2320
13	2221	2173	2208	2209	2185	2214	2289	2273	2288	2247	2119
14	2065	2003	2046	2029	2029	2059	2134	2115	2131	2101	1998
15	1918	1844	1906	1875	1895	1909	1999	1957	1985	1963	1851
17	1697	1622	1680	1675	1676	1703	1776	1725	1746	1732	1631
18	1602	1539	1589	1589	1578	1599	1663	1622	1652	1632	1557
20	1445	1384	1411	1423	1423	1445	1524	1456	1476	1501	1407
25	1157	1075	1163	1153	1117	1143	1232	1159	1188	1196	1141

scenario. However, the numbers are still close to each other so aggregating the data over multiple runs of the same scenario should yield meaningful data.

6.3.2 Reliability

To find out whether the network works reliably, we first needed to ensure the created primary network structure is actually a Delaunay graph. We checked the neighbourhood views among all neighbours are consistent so nodes are actually able to receive messages. As long as messages are in transit, the neighbourhood state might not be consistent. Therefore, we did not allow node join or leave in the last minute of the eight hour runs. This minute is more than enough to reach a consistent state considering the maximum message latency. After that, we recorded the state of the network.

Furthermore, we were also interested in the sizes of neighbourhood sets and routing tables. The former allows to estimate how even the user traffic will be distributed and how tightly a node is integrated in the network. The routing table sizes show whether the overlay can actually provide disjoint paths that are reasonably short to contain the influence of malicious nodes. Both together show the routing state

Table 6.2: Number of running nodes at the end of each run

IAD	\bar{n}	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
6	1169	1218	1197	1129	1177	1181	1133	1159	1157	1183	1153
7	1015	1019	1019	1040	1017	985	1026	1029	978	1021	1019
8	893	930	921	923	863	864	897	908	856	880	888
9	788	836	813	796	754	782	768	791	762	772	801
10	707	716	726	709	696	669	743	680	692	683	751
11	639	660	651	642	653	613	668	595	625	634	649
12	593	616	593	624	582	560	620	527	622	608	577
13	546	549	563	587	539	529	538	531	568	546	506
14	507	498	525	514	496	498	487	513	519	523	497
15	465	460	470	456	472	442	460	462	504	493	426
17	409	408	416	406	414	410	411	406	423	429	364
18	384	383	387	390	376	372	384	383	400	404	357
20	346	340	326	340	355	334	361	335	344	397	324
25	276	241	283	282	270	264	288	266	296	299	266

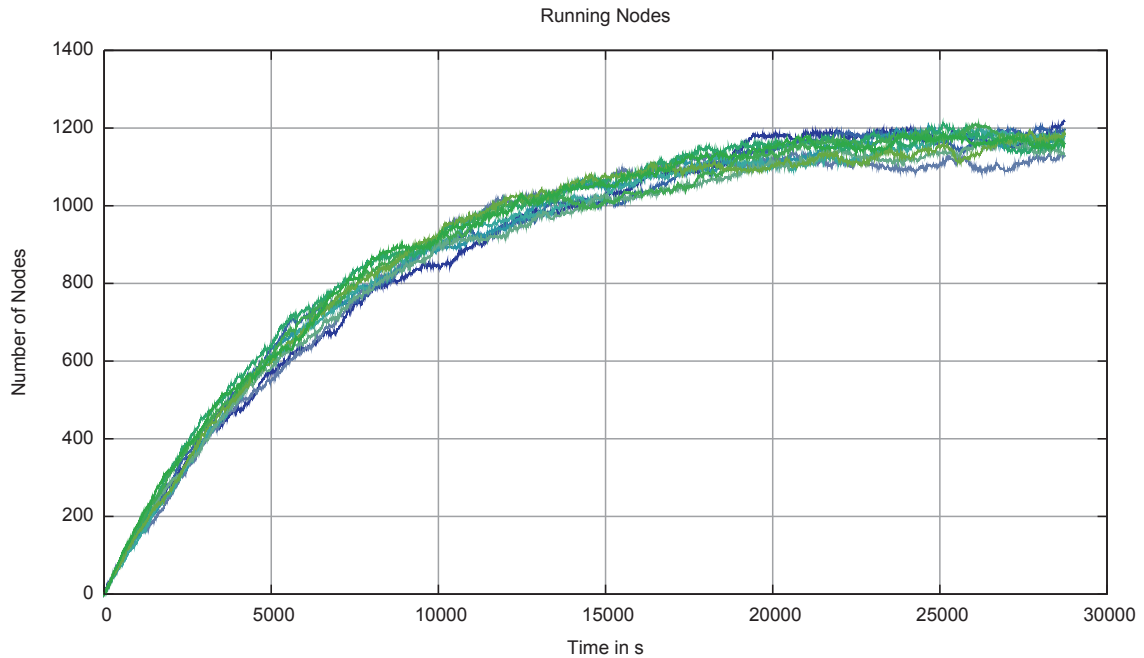
**Figure 6.3:** Number of nodes in the network with $E(SLD) = 2h$ and $E(IAD) = 6s$

Table 6.3: Neighbourhood Sizes Evaluation $r = 4$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	5.9601	1.3244	0.0045	0.0369	3	0.0000	14	0.9487
7	1015	5.9566	1.3746	0.0058	0.0310	3	0.0000	14	1.0750
8	893	5.9497	1.3806	0.0059	0.0383	2	0.3162	13	0.5270
9	788	5.9484	1.3766	0.0087	0.0446	3	0.0000	15	1.1738
10	707	5.9379	1.4116	0.0052	0.0527	3	0.0000	14	1.2472
11	639	5.9349	1.4025	0.0050	0.0459	3	0.0000	14	1.3703
12	593	5.9350	1.4207	0.0077	0.0406	3	0.0000	13	0.8233
13	546	5.9249	1.3735	0.0122	0.0323	3	0.0000	12	0.8756
14	507	5.9219	1.4073	0.0105	0.0402	3	0.0000	15	1.2472
15	465	5.9191	1.3843	0.0112	0.0389	3	0.0000	13	0.8756
17	409	5.9068	1.4080	0.0133	0.0474	3	0.0000	13	0.6992
18	384	5.9027	1.4334	0.0131	0.0456	2	0.3162	13	1.0750
20	346	5.8942	1.4229	0.0113	0.0518	3	0.0000	14	1.1595
25	276	5.8615	1.4278	0.0189	0.0619	3	0.0000	12	0.6667

and the number of open connections since there will only be a few shortcuts without user traffic.

Primary Structure

In all of our test runs, the neighbourhood views were consistent. We conclude our self-stabilizing algorithm reliably creates a Delaunay graph. The average number of neighbours for the scenarios with $r = 4, 9, 16$ is shown in the tables 6.3, 6.4, and 6.5. The first column IAD shows the average inter arrival time used in the exponential distribution. The second column \bar{n} contains the average number of nodes running at the end of the scenario. The column $\bar{\bar{x}}$ shows the average number of neighbours per run averaged over the ten runs. It varies from 5.86 for the smaller networks to 5.96 for the larger networks. This variation is pretty small and consistent with the property of planar graphs having a maximum average node degree of six. Surprisingly, the number of zones k does not seem to have an influence on the average number of neighbours as the range is nearly identical in the three tables.

Table 6.6 shows the same evaluation performed on random graphs of sizes 250, 500, and 1000. These sizes are similar to the sizes created in the IAD25, IAD14, and IAD7 scenario. Comparing their average node degree to the respective scenarios

Table 6.4: Neighbourhood Sizes Evaluation $r = 9$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	5.9634	1.3789	0.0034	0.0302	2	0.3162	15	1.3333
7	1015	5.9559	1.3878	0.0023	0.0367	3	0.0000	16	1.2293
8	893	5.9527	1.4095	0.0037	0.0363	3	0.0000	14	0.9189
9	788	5.9507	1.4374	0.0067	0.0440	2	0.3162	14	1.1005
10	707	5.9410	1.4408	0.0081	0.0499	3	0.0000	14	0.9944
11	639	5.9400	1.4333	0.0086	0.0189	2	0.3162	15	1.3166
12	593	5.9310	1.4461	0.0136	0.0378	3	0.0000	16	1.4181
13	546	5.9236	1.4792	0.0077	0.0492	3	0.0000	14	0.9487
14	507	5.9174	1.4778	0.0109	0.0607	3	0.0000	14	1.0593
15	465	5.9095	1.4780	0.0069	0.0452	2	0.3162	14	0.9661
17	409	5.9020	1.4515	0.0103	0.0590	2	0.3162	14	1.1353
18	384	5.8992	1.4711	0.0149	0.0748	3	0.0000	13	1.0750
20	346	5.8967	1.4994	0.0111	0.0596	3	0.0000	13	0.6325
25	276	5.8719	1.4958	0.0112	0.0484	3	0.0000	13	0.9189

Table 6.5: Neighbourhood Sizes Evaluation $r = 16$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	5.9629	1.4018	0.0044	0.0268	2	0.3162	14	0.9718
7	1015	5.9584	1.4270	0.0055	0.0466	3	0.0000	14	0.8433
8	893	5.9517	1.4569	0.0054	0.0452	3	0.0000	14	0.6325
9	788	5.9492	1.4498	0.0073	0.0416	3	0.0000	14	0.7888
10	707	5.9428	1.4406	0.0088	0.0395	3	0.0000	14	1.2293
11	639	5.9393	1.4542	0.0062	0.0547	3	0.0000	14	0.9944
12	593	5.9321	1.4795	0.0083	0.0256	3	0.0000	14	0.9189
13	546	5.9267	1.4755	0.0079	0.0429	3	0.0000	15	1.6465
14	507	5.9164	1.4649	0.0098	0.0662	3	0.0000	12	0.6992
15	465	5.9118	1.4649	0.0118	0.0709	3	0.0000	15	1.2867
17	409	5.9004	1.5112	0.0114	0.0529	3	0.0000	13	0.8433
18	384	5.9049	1.4683	0.0140	0.0694	3	0.0000	13	0.6992
20	346	5.8914	1.4987	0.0145	0.0347	3	0.0000	13	0.6325
25	276	5.8648	1.4905	0.0164	0.0594	3	0.0000	12	0.4830

Table 6.6: Neighbourhood Sizes Evaluation of Random Delaunay Graphs

n	\bar{x}	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
250	5.8624	1.3335	0.0172	0.0451	3	0.0000	11	0.5676
500	5.9260	1.3632	0.0081	0.0398	3	0.0000	12	0.6325
1000	5.9596	1.3654	0.0053	0.0149	3	0.0000	13	0.6992

shows a minimal increase. The difference is bigger in the fourth column \bar{s}_x showing the average standard deviation of the distribution of the number of neighbours in each run. For each run, we calculated the standard deviation using the estimator $s = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$ and averaged it over the ten runs.

For the random graphs, the variation seems to be independent from the number of nodes with a standard deviation of around 1.35. The recorded runs approach this value only in the IAD6 run with the other runs showing a standard deviation close to 1.45. This means smaller test scenarios create a larger variation in the number of neighbours. This is most likely an effect of the distribution of nodes in zones. Nodes at the border of a zone connected to nodes from other zones are more likely to have a higher number of neighbours – especially nodes at the corner of a zone.

This can also be seen in the maximum node degree in the column \max_x . More nodes increase the chance of one of the nodes having a higher maximum node degree in the random graph. This trend seems to exist in our recorded runs as well. However, the maximum number of nodes is two nodes greater. On the other hand, there seems to be a larger variation in the maximum number of nodes among the different runs as can be seen in the last column s_{\max_x} . Thus, the maximum number of neighbours seems to depend more on the distribution picked and is less stable among runs with the same settings.

On the contrary, both the average number of neighbours as well as the standard deviation of neighbours show much less variation among different runs with standard deviations being close to zero as shown in columns $s_{\bar{x}}$ and s_{s_x} .

Finally, all evaluated random graphs and the majority of the test runs feature a minimum node degree of 3. In some test scenarios, there were also single runs with a minimum node degree of 2. These loosely integrated nodes are more likely to have a considerable share of their traffic not being forwarded by malicious neighbours. However, these minimum node degrees can only be created at nodes in the corners of the ID space, so there are only a few nodes integrated that loosely.

Table 6.7: Filling degree of the routing table for $r = 4$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	0.7388	0.0301	0.0022	0.0026	0.5833	0.0000	0.7500	0.0000
7	1015	0.7383	0.0315	0.0023	0.0032	0.5000	0.0439	0.7500	0.0000
8	893	0.7371	0.0320	0.0032	0.0041	0.5000	0.0351	0.7500	0.0000
9	788	0.7374	0.0322	0.0032	0.0041	0.5000	0.0351	0.7500	0.0000
10	707	0.7355	0.0334	0.0052	0.0049	0.5000	0.0264	0.7500	0.0000
11	639	0.7344	0.0357	0.0023	0.0024	0.5000	0.0403	0.7500	0.0000
12	593	0.7303	0.0397	0.0045	0.0040	0.5000	0.0430	0.7500	0.0000
13	546	0.7308	0.0389	0.0031	0.0029	0.5000	0.0403	0.7500	0.0000
14	507	0.7288	0.0403	0.0039	0.0030	0.5000	0.0264	0.7500	0.0000
15	465	0.7292	0.0403	0.0050	0.0045	0.5000	0.0439	0.7500	0.0000
17	409	0.7275	0.0425	0.0039	0.0048	0.4167	0.0583	0.7500	0.0000
18	384	0.7246	0.0437	0.0059	0.0057	0.5000	0.0430	0.7500	0.0000
20	346	0.7250	0.0421	0.0079	0.0053	0.5000	0.0351	0.7500	0.0000
25	276	0.7204	0.0486	0.0063	0.0066	0.5000	0.0351	0.7500	0.0000

To sum up, our algorithm reliably creates Delaunay graphs with properties very similar to those of random Delaunay graphs. The small differences can be attributed to the zoned distribution of IDs in our test runs.

Routing Table

Routing tables need to be filled to achieve a satisfying routing performance. The more entries are filled, the shorter paths become. It is especially important the top layer of routing tables is filled. This layer allows the direct far jump into the different zones and is necessary to realize disjoint paths. Therefore, we measured filling degrees of routing tables in the different scenarios.

The filling degrees of the routing tables for replication factors $r = 4, 9, 16$ are shown in tables 6.7, 6.8, and 6.9. The number of routing table layers was fixed at four in all test scenarios.

With $r = 4$, the routing table offers $4 * 3 = 12$ entries in total. The average filling degree $\bar{\bar{x}}$ is close to the maximum filling degree \max_x which is 75% equalling 9 filled entries in all scenarios. This is actually the maximum possible degree with our

distribution of IDs in the key space as defined in section 4.5.4. It causes the second routing table layer to be completely empty as no nodes are in the sub-zones targeted by this layer. Only three of the four layers can theoretically be filled. It is important to note that this does not have a negative effect on the expected routing performance. The second layer being empty means the routing cannot directly jump into the sub-zones of this layer. Routing there would only be possible by going through the current sub-zone. However, there are no nodes in these sub-zones so jumping there is simply not needed. Target nodes are either in the current sub-zone or in another top-level zone which could all be reached efficiently by using layer three and below or the top layer. Thus, a filling degree of 75% actually means perfect $O(\log_4 n)$ routing performance can be expected.

The average filling degree $\bar{\bar{x}}$ varies from 72.04% for the IAD25 scenario with 276 nodes on average to 73.88 for the IAD6 scenario with 1169 nodes on average. At the same time the average standard deviation \bar{s}_x shrinks from 4.86% to 3.01%. Thus, more nodes have a filling degree close to the average with increasing network size.

Both the average filling degree $\bar{\bar{x}}$ as well as the average standard deviation \bar{s}_x do not fluctuate a lot among the different runs as can be seen from columns $s_{\bar{\bar{x}}}$ and s_{s_x} . The minimum filling degree \min_x is around 50% in most runs and does not vary much among runs as shown in column s_{\min_x} . With $r = 4$, the routing table is nearly filled completely even in the small scenario with only 276 nodes. There are only a few nodes deviating from this value and at least 50% of the tables are filled in all except one scenario with a minimum degree of 41.67%. Due to the very good overall filling degrees of routing tables, we can expect a $O(\log_4 n)$ routing performance.

We also performed the same evaluation on the top layer of the routing table to find out how well disjoint paths routing will work. In seven out of 14 scenarios, the average filling degree was 100%. The other seven scenarios had an average filling degree between 99.83% and 99.99%. This means only a few nodes were not able to fill their routing table completely in some of the runs.

Inspecting the cause of this, we discovered a limitation of the connection management. In conjunction with a certain constellation of operations, it can lead to nodes closing routing table connections leaving entries empty until they are refilled automatically. This happens when node a opens a shortcut connection to node b and b signals a it is not interested in the connection by sending a `ConnectionInfo` to a . Then node c in the routing table of b fails and a is the best replacement for c . Then, a is added to b 's routing table while a has still stored b does not need the connection. When a performs its shortcut maintenance and discovers it no longer needs the

Table 6.8: Filling degree of the routing table for $r = 9$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	0.6957	0.0370	0.0021	0.0025	0.4688	0.0428	0.7500	0.0000
7	1015	0.6825	0.0375	0.0034	0.0023	0.5313	0.0257	0.7500	0.0000
8	893	0.6728	0.0395	0.0046	0.0020	0.5000	0.0211	0.7500	0.0000
9	788	0.6600	0.0415	0.0043	0.0020	0.5000	0.0211	0.7500	0.0000
10	707	0.6492	0.0432	0.0053	0.0042	0.5000	0.0151	0.7500	0.0132
11	639	0.6406	0.0446	0.0038	0.0024	0.5000	0.0151	0.7500	0.0151
12	593	0.6342	0.0444	0.0066	0.0026	0.4688	0.0147	0.7500	0.0151
13	546	0.6234	0.0435	0.0058	0.0023	0.4375	0.0221	0.7500	0.0099
14	507	0.6200	0.0454	0.0049	0.0037	0.4375	0.0264	0.7500	0.0177
15	465	0.6108	0.0439	0.0046	0.0046	0.4688	0.0151	0.7500	0.0198
17	409	0.5988	0.0434	0.0056	0.0043	0.4375	0.0257	0.7188	0.0198
18	384	0.5935	0.0421	0.0057	0.0033	0.4375	0.0198	0.7188	0.0147
20	346	0.5829	0.0431	0.0041	0.0041	0.4375	0.0198	0.7188	0.0231
25	276	0.5638	0.0412	0.0076	0.0059	0.4375	0.0161	0.6875	0.0295

shortcut to b , it will close the connection to b . When b notices the connection close, it will start looking for a new routing table entry and it will rediscover a and add it again. This time, a already removed the stored ConnectionInfo and the connection will stay open.

To fix this, b needs to send out a positive ConnectionInfo leading to a removing the stored negative interest of b . However, this would still leave a short time window during which a could still close the connection while the positive ConnectionInfo is in transit. Furthermore, it would increase the number of exchanged ConnectionInfos considerably. Finally, this constellation becoming visible is due to our scenario. The system runs with joining nodes causing shortcut connections being opened when filling the routing tables and requesting their own areas. Then suddenly before the end of the simulation the churn stops and the system runs idle for a minute which happens to be close to the shortcut timeout. Thus, shortcut connections are closed because no additional traffic is generated. In practice, shortcuts will be kept open much longer because of the ongoing user traffic.

With $r = 9$, the routing table offers $4 \times 8 = 32$ entries in total. The maximum possible filling degree is again 75% equalling 24 filled entries because of the distribution of IDs. With 1169 nodes on average in the IAD6 scenario, the average filling degree

\bar{x} is quite close to this theoretical maximum with 69.57%. With fewer nodes, the routing table becomes more empty down to 56.38% in the IAD25 scenario with 276 nodes on average. However, this is still about 75% of the theoretically obtainable maximum.

The average standard deviation of filling degrees \bar{s}_x is quite similar to $r = 4$ growing from 3.7% at the IAD6 scenario to 4.12% at the IAD25 scenario. Fewer nodes introduce more variation as the ID space is filled less uniformly. The standard deviations of average node degrees $s_{\bar{x}}$ and standard deviations s_{s_x} are very low showing the results to be consistent among the runs.

The minimum filling degree \min_x varies from 43.75% to 53.13% showing a trend to rise with increasing node numbers. Against this trend, a filling degree of 46.88% is realized at the scenario with most nodes. The considerably higher standard deviation s_{\min_x} of 4.28% in this scenario suggests there is just a single outlier run in this scenario and the minimum node degree was higher in the other runs following the trend. The maximum filling degree \max_x of 75% is reached only for scenarios with at least 465 nodes on average. For all runs in a scenario to reach that maximum degree, there actually need to be at least 788 nodes on average as shown by the standard deviation of maxima s_{\max_x} .

The strong correlation between the number of nodes and the average, minimum, and maximum filling degree suggests the routing table maintenance mechanism reliably fills routing tables to the possible extent. If an entry is empty, this is most likely due to the fact that there is no node matching the entry constraints. The more routing table entries are used for the same ID space, the tighter the constraint on each entry become. With $r = 9$, the number of routing table entries is large enough to decrease the average filling degree considerably for scenarios with a lower number of nodes.

On the other hand, since the filling mechanism fills to the possible extend, the lower filling degree may not actually have a negative impact on routing performance. The entries that cannot be filled are in the lower layers with the tighter constraints. If they are not filled, the node density is too low to fill them. If the node density is low, a jump in the low layer bridging a certain ID space distance would just not save hops compared to using the neighbourhood for routing. If the distances to neighbours are as big as the distances to entries on a routing table layer, that layer is just not needed to improve performance.

Although the filling degree is smaller, the routing performance with $r = 9$ would still be better than with $r = 4$ since the routing performance is $O(\log_9 n)$. However, in

Table 6.9: Filling degree of the routing table for $r = 16$

IAD	\bar{n}	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\min_x	s_{\min_x}	\max_x	s_{\max_x}
6	1169	0.4756	0.0509	0.0040	0.0023	0.3333	0.0105	0.6167	0.0136
7	1015	0.4652	0.0511	0.0034	0.0021	0.3167	0.0118	0.6167	0.0141
8	893	0.4541	0.0495	0.0032	0.0031	0.3167	0.0095	0.6000	0.0158
9	788	0.4424	0.0476	0.0028	0.0038	0.3167	0.0118	0.5833	0.0196
10	707	0.4343	0.0459	0.0039	0.0027	0.3000	0.0131	0.5833	0.0192
11	639	0.4246	0.0442	0.0050	0.0043	0.3000	0.0111	0.5500	0.0157
12	593	0.4186	0.0447	0.0042	0.0028	0.3000	0.0123	0.5500	0.0196
13	546	0.4126	0.0441	0.0050	0.0040	0.3000	0.0086	0.5667	0.0211
14	507	0.4072	0.0433	0.0047	0.0046	0.3000	0.0086	0.5333	0.0131
15	465	0.4018	0.0433	0.0075	0.0054	0.3000	0.0086	0.5500	0.0192
17	409	0.3899	0.0383	0.0048	0.0041	0.3000	0.0070	0.5167	0.0189
18	384	0.3854	0.0372	0.0044	0.0043	0.3000	0.0053	0.5333	0.0299
20	346	0.3784	0.0361	0.0048	0.0038	0.2833	0.0079	0.5167	0.0225
25	276	0.3634	0.0322	0.0042	0.0030	0.2833	0.0053	0.4833	0.0183

scenarios with low node numbers there will be a lot of wasted messages as redundant RtRequests will be sent out and answered that will not fill an entry in the end.

For the top layer, results were similar to the results achieved with $r = 4$. The top layer was nearly always filled completely. The rare exceptions can probably be attributed to the connection management issue involving shortcuts explained before.

Finally, with $r = 16$, a total of $4 * 15 = 60$ routing table entries are available to be filled. The theoretical maximum filling degree is slightly higher than 75% as the second routing table layer might contain up to three nodes at maximum. The 2x2 top left of the 4x4 entries in that layer cover the used part of the ID space. One of these will be empty because it is the part covered by the lower layers. The other three entries might be filled leading to a maximum filling degree of 80%. This maximum filling degree is not even closely reached in any of the scenarios. The maximum filling degree \max_x grows from 48.33% to 61.67% as average network sizes go from 276 nodes to 1169 nodes. Variation among runs is very small for the maximum s_{\max_x} as well as the minimum filling degree s_{\min_x} . The minimum filling s_{\min_x} degree varies between 28.33% and 33.33%, so there is not a lot of improvement even if the average network size grows by a factor of more than four.

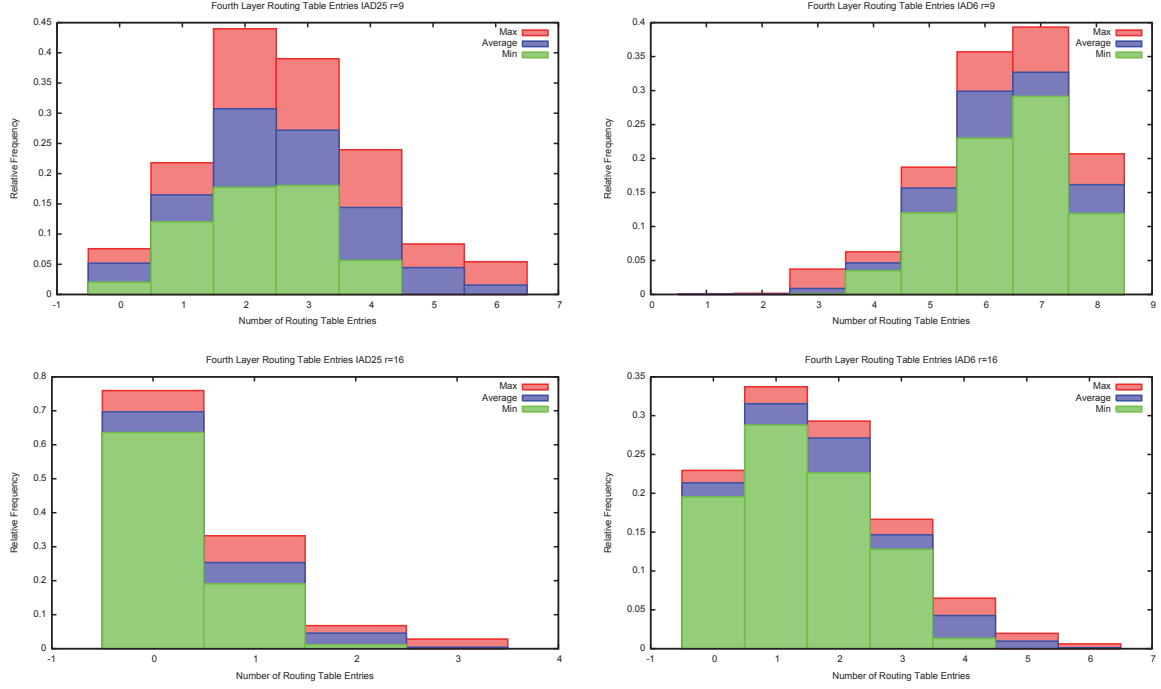


Figure 6.4: Number of fourth layer entries in scenarios IAD6 and IAD25 with $r = 9, r = 16$

The average filling degree $\bar{\bar{x}}$ also only improves from 36.34% to 47.56% with values being consistent among different runs. Surprisingly, the average standard deviation \bar{s}_x actually grows as network sizes grow. This also happens consistently among the runs. We assume the scenarios are still very far away from high filling degrees. Then, small network sizes provide bad filling degrees for all nodes – with small variation. Higher node numbers provide a better average filling degree $\bar{\bar{x}}$ but with a bigger variation \bar{s}_x . We did not reach node numbers high enough to increase the average node degree close to the maximum lowering the variation around the average.

The filling degree of the top layer was nearly always 100% as before for $r = 4$ and $r = 9$ and exceptions are most likely due to the same connection management issue.

The replication factor $r = 16$ will provide the best routing performance with $O(\log_{16} n)$. The low filling degree can mostly be attributed to layer four which would not provide increased performance compared to using the neighbourhood set for routing. This can also be seen by comparing the distribution of the number of nodes for $r = 9$ and $r = 16$ in the IAD6 and the IAD25 scenario in figures 6.4.

With $r = 9$, on average 55% of all nodes have two or three out of eight filled entries in the IAD25 scenario. Thus, the average number of filled entries is around 2.5

contributing a filling share of 31.25%. In the same scenario with $r = 16$, the average number of filled entries is about 0.3 out of 15 entries resulting in filling share of 2%. In the IAD6 scenario, the average number of filled entries with $r = 9$ is about 6.5 resulting in a filling share of 81.25% on the fourth layer. With $r = 16$, there are only about 1.4 filled entries resulting in an average filling share as low as 9%. Therefore, using routing tables with $r = 16$ and three layers would probably be sufficient for our network sizes and save maintenance messages. At the same time, using four layers seems to be reasonable for larger network sizes and $r = 9$.

In a second set of test runs performed while evaluating the storage, we started 200, 400, 600, 800, and 1000 nodes with replication factors $r = 4, 9, 16$ with ten runs each as described in section 6.4.1. Afterwards, we evaluated how many routing table entries actually contained the nodes globally closest to an entry's target position to find out how well the passive filling updates the routing tables. Furthermore, we counted how many entries fulfil our claim of being symmetric. The results are shown in table 6.10.

With $r = 4$, 88% of all filled entries actually contain the node closest to the entry's target position. This value increases up to 97% with $r = 16$ and 200 nodes. As the replication factor increases, RtRequest messages take more different paths to reach a target position. Therefore, more nodes have a chance to passively update their entry when new nodes join. For $r = 9$ and $r = 16$, the ratio decreases slightly with rising node number. This seems to be connected to the number of nodes per zone, which is highest with a low replication factor and a high total number of nodes. The more nodes per zone, the more nodes have a chance to actually miss the RtRequest message of the joining node passing through that zone they would have needed to update the corresponding entry. Nevertheless, about 90% of filled entries actually being closest means the passive filling works well enough considering no effort is needed to regularly refresh routing table entries.

The percentage of symmetric entries is actually below our expectations. With $r = 4$, only about 45% of entries are symmetric. This increases to around 56% with $r = 9$ and 57% with $r = 16$. It seems the chance for an entry to be symmetric are around equal to the chance that another node next to the entry is the one referring to the original node as shown in 4.15. However, the way the overlay is used by the storage leads to shortcut connections being established so nodes contained in routing tables still know the referring nodes.

Table 6.10: Symmetric entries and entries closest to target position averaged over ten test runs

n	r	Filled Entries	Symmetric Entries	Symmetric/Filled	Closest Entries	Closest/Filled
200	4	1,718	788	0.46	1,503	0.87
400	4	3,507	1,556	0.44	3,076	0.88
600	4	5,303	2,330	0.44	4,656	0.88
800	4	7,111	3,160	0.44	6,236	0.88
1000	4	8,902	4,003	0.45	7,822	0.88
200	9	3,452	1,939	0.56	3,112	0.90
400	9	7,669	4,276	0.56	6,903	0.90
600	9	12,179	6,841	0.56	10,959	0.90
800	9	16,994	9,685	0.57	15,208	0.89
1000	9	21,861	12,476	0.57	19,512	0.89
200	16	4,183	2,284	0.55	4,075	0.97
400	16	9,372	5,287	0.56	8,962	0.96
600	16	15,173	8,621	0.57	14,303	0.94
800	16	21,383	12,269	0.57	19,924	0.93
1000	16	27,888	16,102	0.58	25,814	0.93

6.3.3 Scalability

We wanted to find out whether the network maintenance mechanism allows scaling the network to large sizes with a thousand nodes. This would be the case if the total traffic in the network grew linearly with the network size. Then, the average traffic per node would be constant and adding new nodes to the network would not increase load on existing nodes. In addition to that, the load should be distributed evenly so no single node has to process a considerably higher amount of traffic than the rest. We were also interested in how the traffic is distributed between the messages for the primary structure maintenance, the connection management, and the routing table maintenance.

Total Traffic

Figure 6.5 shows the total traffic in the network on the link layer. We recorded the size of every message sent on the link layer on every node and aggregated the sizes for each run. Every point in the figure shows the total traffic and the number of nodes that have been started in a run. Using linear regression, we fitted our measurements to create linear functions. The results suggest the overlay maintenance is in fact nearly perfectly scalable as the traffic grows linearly with the number of nodes that have been in the network during the tests.

Bandwidth per Node

We validated this result by evaluating the average bandwidth per node as shown in figure 6.6. To calculate the bandwidth, we recorded the upstream link layer traffic on each node and averaged it over the node's lifetime. The results show the average bandwidth stays nearly constant or increases only very slightly as network sizes increase. The load on each node is independent of the network size meaning the network maintenance is scalable. Higher replication factors increase the consumed bandwidth as expected. However, on average the required bandwidth is far away from saturating even today's DSL connections.

Figure 6.6 also shows some outliers where the bandwidth requirements are considerably higher than on the other runs of the same scenario. The outliers occur on identical run settings for $r = 4, 9, 16$. At the same time, the total traffic in figure 6.5 does not show any outliers. This means these outliers must be due to short lifetimes leading to increased bandwidth. The settings of these runs probably caused

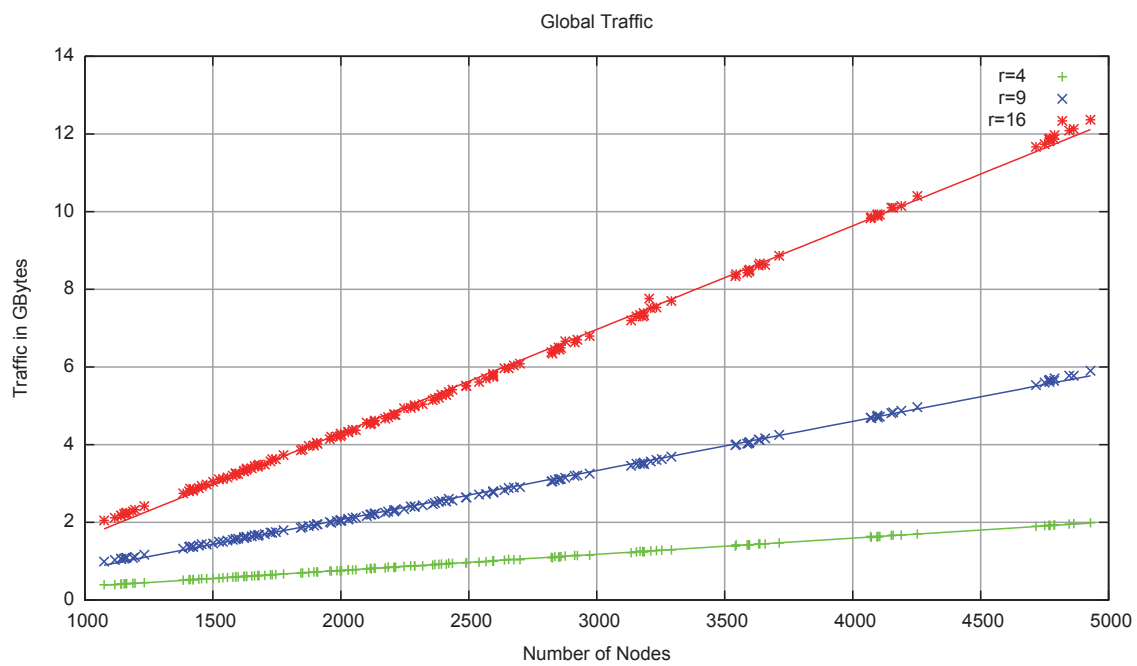


Figure 6.5: Global link layer traffic depending on the number of started nodes

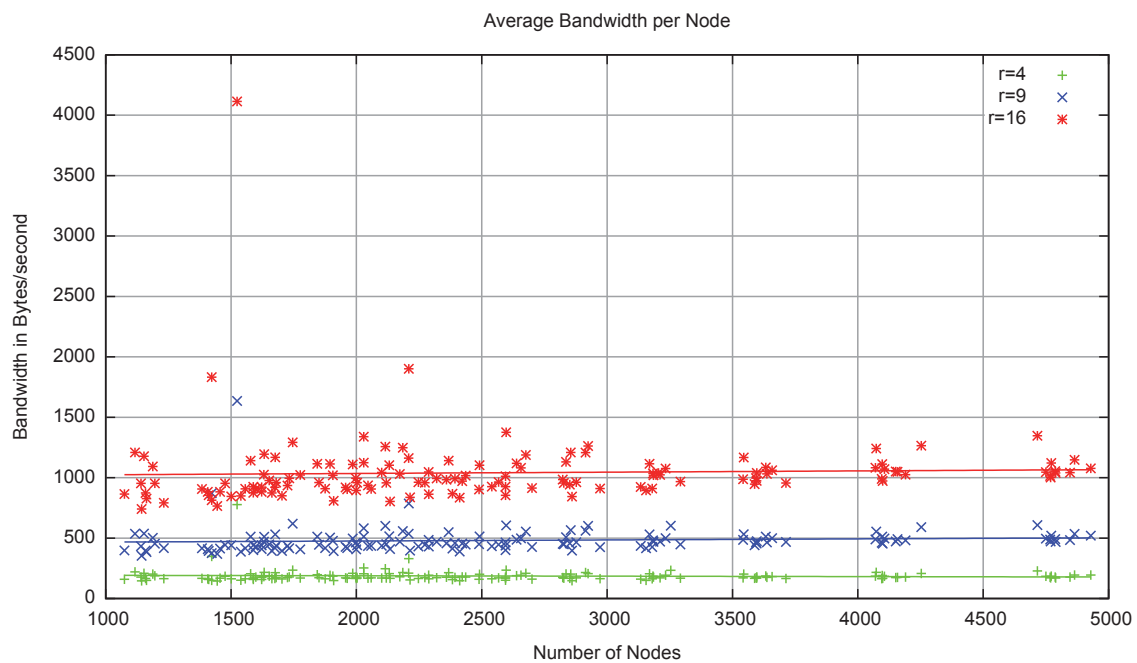


Figure 6.6: Average bandwidth per node depending on the number of started nodes

some nodes to leave right after joining still performing the expensive join procedure. The much higher bandwidth on these nodes caused the heavy bias on the average bandwidths.

Distribution of Message Types

Whenever the overlay forwarded a message to another node, we recorded the type and the size of the message payload. We aggregated this information over all nodes for each run and then averaged the frequencies of the resulting frequency distributions. Figure 6.7 shows the absolute numbers and shares of each message type on the total payload traffic for the scenario IAD6 and $r = 4$. The column *Payload Traffic* considers only pure payload sizes as they could be recorded on the overlay layer. The link layer is payload agnostic so recording traffic there does not allow to differentiate between different payload types. The column *Messages* shows the share of the total number of messages for each message type. The column *Gross Traffic* estimates the real traffic on the link layer by adding the size of the header information to each recorded payload. Figure 6.8 shows the same information for $r = 9$ and figure 6.9 shows it for $r = 16$.

For $r = 4$, the net payload traffic sums up to 0.9 GByte. The majority of data are NetworkViews for the Delaunay structure maintenance and RtRequests for the routing table maintenance each accounting for around 42% of the net traffic. Connection management messages and RtResponses each make up another 7% of the traffic while JoinRequests represent only 2% of the payload traffic. Considering the number of messages, RtRequests nearly make up two thirds of all messages and the number of RtResponses and ConnectionInfos is around equal to the number of NetworkViews. The NetworkView and RtRequest shares of net traffic being equal is due to the much larger size of NetworkViews. This also explains the increase in shares of RtRequests, RtResponses, ConnectionInfos, and JoinRequests in the gross traffic column. The additional header information has more effect on small messages, resulting in 1.25 GByte gross traffic. In total, the routing table maintenance accounts for half of the gross traffic for $r = 4$.

With $r = 9$, net traffic increases significantly to 2.8 GByte while gross traffic grows even more to 4.3 GByte. The larger difference can be explained by the small messages of routing table maintenance making up for the vast majority messages and traffic. The number of NetworkViews stays the same while the number of routing table maintenance messages grows to make up 90% of the total number of messages. The number of ConnectionInfos also grows but not as strong as RtRequests and

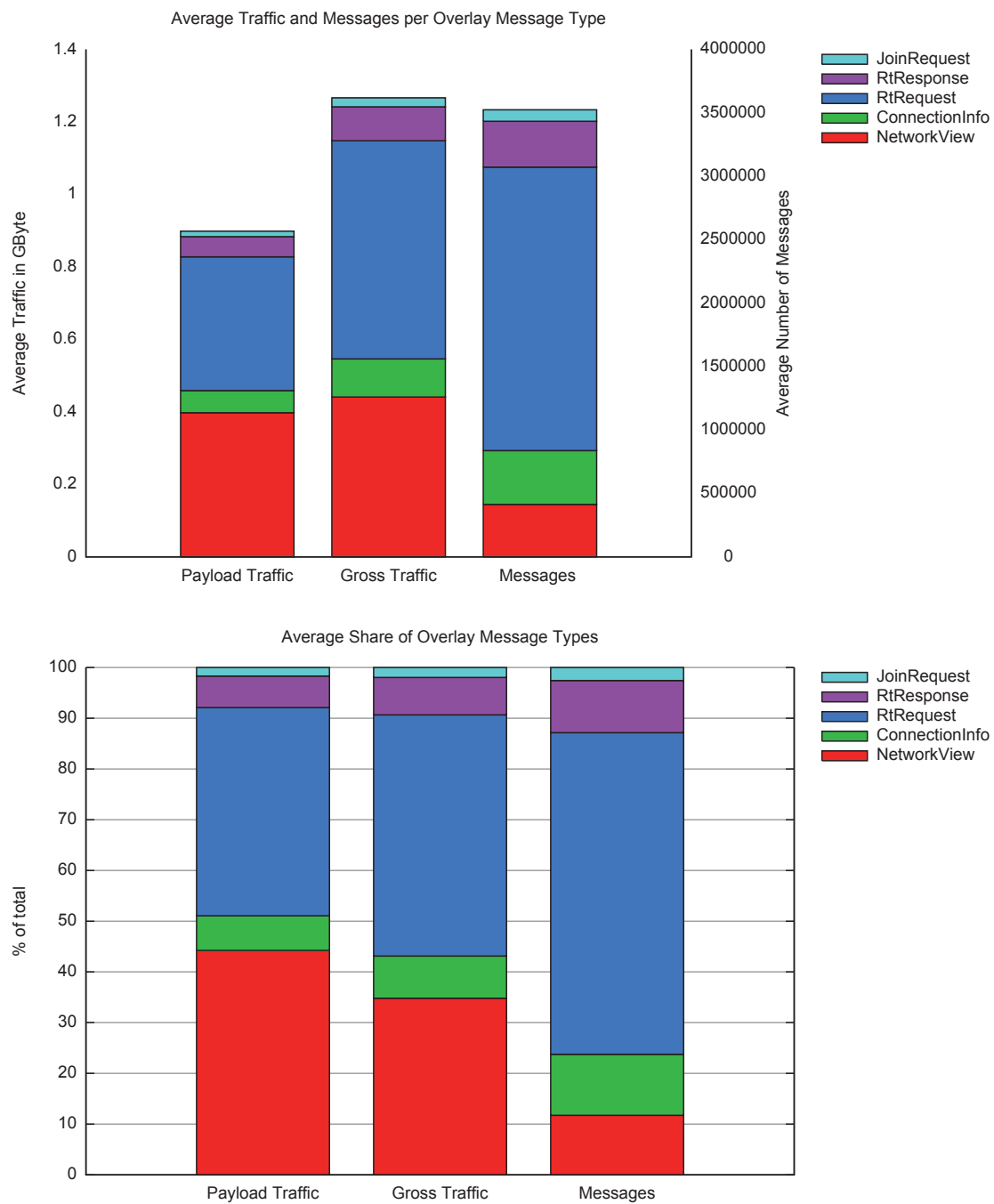


Figure 6.7: Frequency distribution of message types for scenario IAD6, $r = 4$

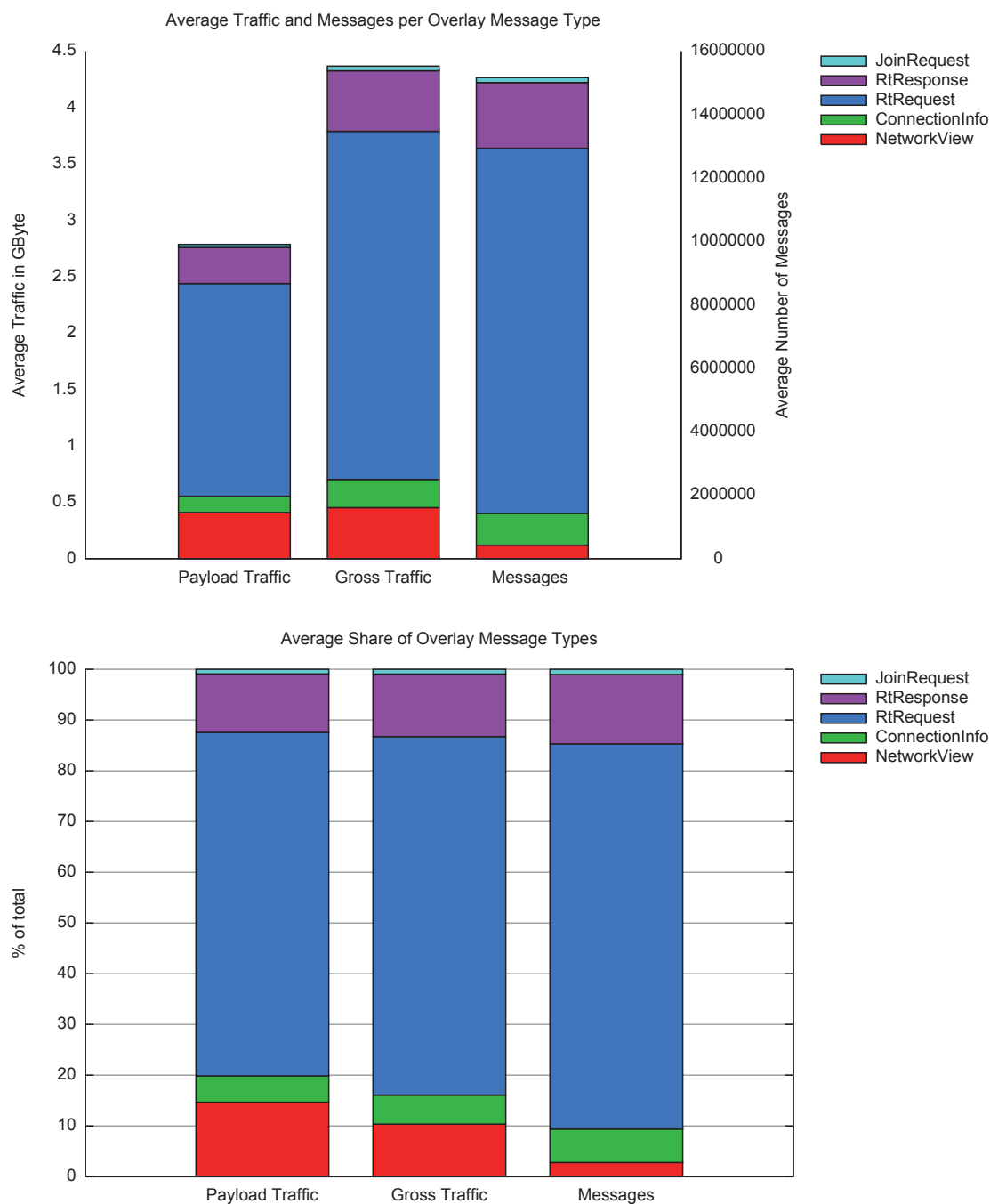


Figure 6.8: Frequency distribution of message types for scenario IAD6, $r = 9$

RtResponses so its relative share still shrinks. ConnectionInfos are sent out before connections are closed so its number is related to the number of connections growing linearly with the routing table size. The number of RtRequests and RtResponses grows proportionally to the square of the routing table size due to the r redundant request paths for each entry.

Therefore, routing table maintenance completely dominates messages and traffic for $r = 16$. It makes up 90% of the gross traffic and 95% of the number of messages. Due to the majority of messages being very small, the difference between net traffic and gross traffic is even bigger with net traffic accounting for 6.3 GByte in total and gross traffic accounting for 10.5 GByte. This number is still smaller than the ca. 12 GByte total traffic that have been recorded in the IAD6 scenario shown in figure 6.5. This difference can be explained considering connection establishment, certificate exchange, and disconnection messages that are also part of the total traffic.

6.3.4 Resilience

To be resilient against attackers, the overlay must not allow malicious nodes to occupy a more than proportional share in the routing state of honest nodes. Therefore, we used test scenarios including malicious nodes and evaluated how often these nodes occur in other nodes' neighbourhood sets and routing tables.

Scenario Setup

Malicious nodes will not just randomly join the network and tamper with the routing of messages. They will act in a coordinated way and try to create a worst-case scenario for the overlay. Their chances to gain influence are best when the number of nodes in the network is still very low and the network is growing. A certain minimum number of honest nodes are necessary to start up the overlay. Otherwise malicious node could just eclipse every joining node. Therefore, we assume an attacker will let his malicious nodes join directly after the minimum amount of honest nodes are in the network and it got opened to other nodes. Then, the malicious nodes join and try to obtain more influence.

In our test scenarios, we first started up 100 honest nodes and then 5, 10, 20, and 30 malicious nodes. The share of malicious nodes after the last malicious node joined is 4.8%, 9.1%, 16.7%, and 23.1%, respectively. Controlling 30 malicious nodes paying

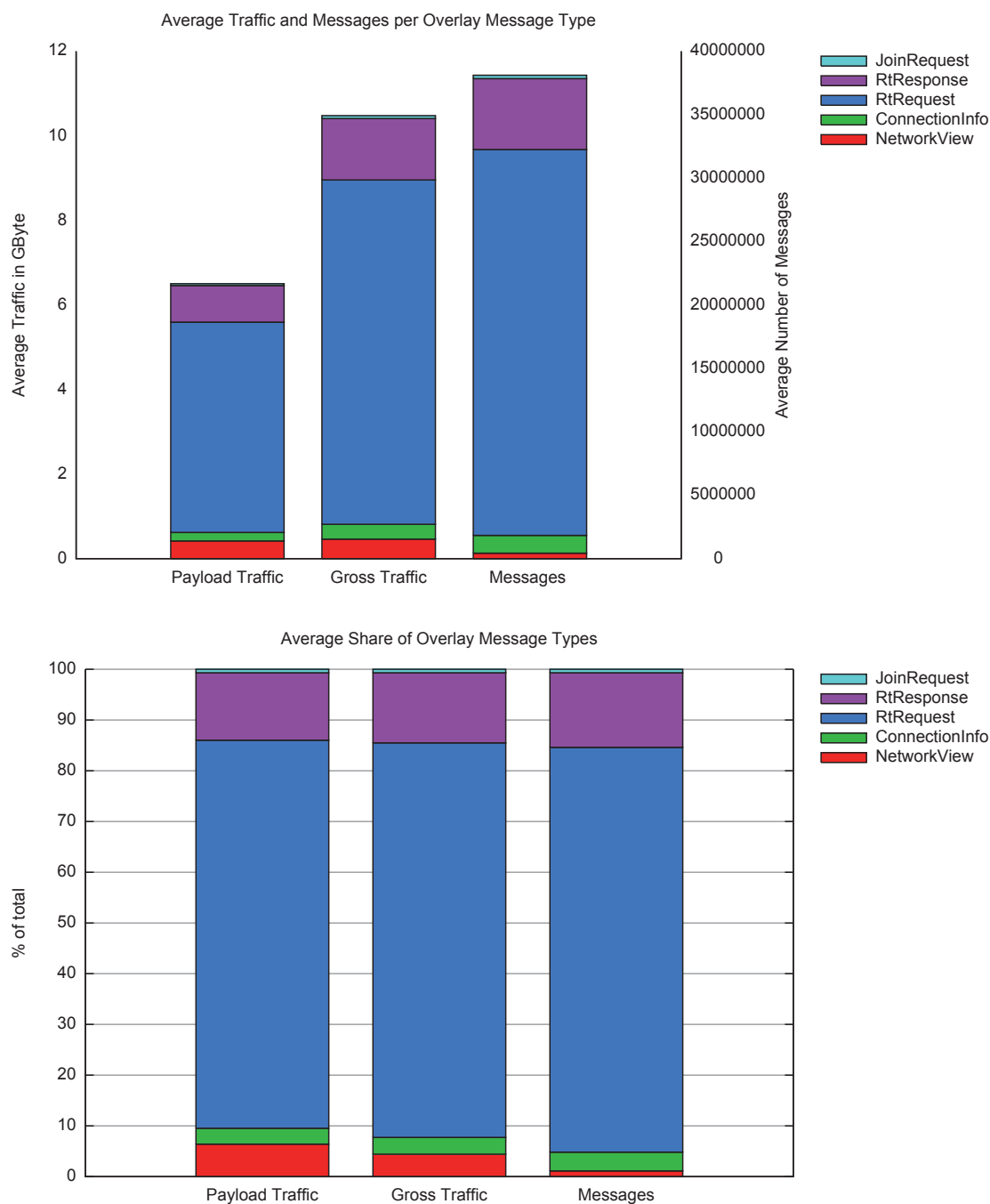


Figure 6.9: Frequency distribution of message types for scenario IAD6, $r = 16$

for 30 identities is actually quite an effort for an attacker. However, in a second set of test scenarios we started 200 honest nodes and afterwards 10, 20, 40, and 60 malicious nodes, increasing the attacker power and honest nodes numbers to obtain the same ratio of honest to malicious nodes. By increasing the minimum number of honest nodes, the resulting share of malicious nodes can always be toned down at the price of having to supply more honest nodes initially. Once the network contains thousands of nodes, an attacker will have a much harder time trying to increase his influence.

We used the IAD6 scenario to let the network grow to around 1200 nodes and performed ten test runs with the replication factors $r = 4, 9, 16$ totalling 240 runs. The test runs were exactly the same as in the reliability and scalability tests so nodes will join and leave at exactly the same time with one exception. Malicious nodes will not leave the network as the attacker will try to keep any influence he already gained.

Malicious Node Behaviour

A malicious node can show any kind of malicious behaviour. However, we have to define how the malicious node acts taking the countermeasures in place into account. Depending on the countermeasures, the malicious behaviour can be grouped into multiple categories. First, there is misbehaviour where countermeasures are targeted at making its success very unlikely, effectively preventing it. This is generally achieved using redundancy. Wrong routing of messages is an example for this kind of behaviour as we use redundant paths to compensate for wrong routing but do not try to detect it.

Second, there is misbehaviour that can only be detected and punished. If it is easy to detect and punish because proof of misbehaviour can be presented to the CA, malicious nodes will not show this behaviour if they cannot gain much advantage from doing it. An example for this is sending out NetworkViews with nodes currently not in the network to destabilize other nodes. Therefore, we do not include this type of malicious behaviour in the evaluation.

If detection and punishment only work by raising disputes at the CA using sophisticated detection algorithms with certain thresholds to identify malicious nodes, implementing this kind of malicious behaviour in the evaluation will not create much insight because it is just an arms race. If the malicious nodes know the thresholds

and the detection algorithms, they can always stay just below the thresholds to remain undetected. Instead of reporting how many malicious nodes have been caught and excluded, the thresholds will just permit tampering a certain number of times before punishment occurs.

Sending out wrong NetworkViews suppressing neighbours is an example for this kind of malicious behaviour. When to use this attack is actually very hard to decide. It is only useful when messages do not directly pass malicious nodes but when receiving the message can be delayed by destabilizing the receiver. The attacker needs to know or predict the flow of messages passing nodes not under his control. Considering the fact that an attacker can only slightly increase its influence at a considerable risk compared to just tampering with message routing and the difficulties in modelling the attacker behaviour, we also did not include this type of attack.

Finally, we also did not implement attacks like eclipsing that can only really succeed if all nodes received from the bootstrapper are malicious due to the countermeasures in place. We only counted the number of malicious nodes received from the bootstrapping service and regarded a node that received malicious nodes only as being eclipsed.

We only included attacks that are dealt with using redundancy: attacks on routing tables. Since no detection and punishment takes place for this kind of attack, we assume malicious nodes very aggressively answer every RtRequest message coming their way with an RtResponse themselves and do not forward the RtRequest. This way, they try to increase their share in routing tables to the maximum.

Routing Table Shares of Malicious Nodes

In the test scenarios, a group of malicious nodes joins the overlay after a minimum number of honest nodes have joined. At this time, the malicious nodes have the highest share of nodes in the network. The network continues to grow with honest nodes joining and leaving and the share of malicious nodes shrinks. This should be reflected in the share of malicious nodes in the routing tables. Ideally, it should also shrink and be as low as the total share of malicious nodes in the network.

Table 6.11 shows the shares of malicious nodes in the routing tables of honest nodes aggregated over all runs for a given scenario. The first column identifies the number of honest (H) nodes that were started first followed by the number of malicious nodes (M). The second column m/\bar{n} shows the average share of malicious nodes in

Table 6.11: Share of malicious nodes in routing tables

Scen.	m/\bar{n}	r	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\max_x	$\overline{\max_x}$	s_{\max_x}
H100	0.0043	4	0.0045	0.0218	0.0019	0.0045	0.2222	0.1466	0.0409
M5		9	0.0045	0.0139	0.0012	0.0019	0.1364	0.0946	0.0208
		16	0.0043	0.0119	0.0011	0.0014	0.0909	0.0750	0.0074
H100	0.0085	4	0.0082	0.0294	0.0026	0.0049	0.2500	0.2198	0.0294
M10		9	0.0096	0.0208	0.0020	0.0024	0.1818	0.1333	0.0320
		16	0.0087	0.0168	0.0016	0.0016	0.1200	0.0914	0.0152
H200	0.0085	4	0.0093	0.0317	0.0033	0.0056	0.3333	0.2264	0.0499
M10		9	0.0094	0.0202	0.0016	0.0019	0.1429	0.1166	0.0190
		16	0.0086	0.0174	0.0012	0.0015	0.1379	0.1053	0.0185
H100	0.0168	4	0.0172	0.0431	0.0044	0.0057	0.5556	0.2851	0.1138
M20		9	0.0181	0.0283	0.0023	0.0022	0.1818	0.1618	0.0216
		16	0.0172	0.0240	0.0017	0.0013	0.1600	0.1267	0.0184
H200	0.0168	4	0.0210	0.0476	0.0036	0.0035	0.3333	0.2861	0.0508
M20		9	0.0178	0.0274	0.0022	0.0014	0.2105	0.1558	0.0256
		16	0.0172	0.0242	0.0012	0.0014	0.2069	0.1373	0.0278
H100	0.0250	4	0.0274	0.0547	0.0028	0.0024	0.5000	0.3486	0.0770
M30		9	0.0273	0.0342	0.0024	0.0013	0.2273	0.1973	0.0181
		16	0.0261	0.0291	0.0027	0.0015	0.2400	0.1560	0.0341
H200	0.0331	4	0.0418	0.0672	0.0066	0.0055	0.5556	0.3722	0.0830
M40		9	0.0354	0.0392	0.0028	0.0018	0.3000	0.2287	0.0324
		16	0.0341	0.0343	0.0019	0.0025	0.2308	0.1820	0.0263
H200	0.0489	4	0.0600	0.0804	0.0079	0.0047	0.5556	0.4472	0.0695
M60		9	0.0515	0.0466	0.0041	0.0014	0.3000	0.2598	0.0256
		16	0.0500	0.0413	0.0033	0.0026	0.2500	0.2126	0.0196

the network. The replication factor r used in a test scenario is given in the third column.

The average share of malicious nodes in the routing tables in column \bar{x} is very close to the share of malicious nodes m/\bar{n} in the network throughout all scenarios. Even a replication factor of $r = 4$ is sufficient to keep the 6% average share of malicious nodes in routing tables in the scenario with 60 malicious nodes just slightly above the 4.89% share of malicious nodes in the network. With $r = 9$ and $r = 16$, the shares get even closer to that value with 5.15% and 5%. Increasing the replication factor yields closer results in all scenarios with at least 20 malicious nodes. The more nodes are malicious, the more likely they can tamper successfully. Increasing redundancy to prevent this has a visible effect only when there are enough malicious nodes.

Higher replication factors also lead to the standard deviation of malicious nodes' share to decrease as shown in column \bar{s}_x . Not only the average share shrinks, there is also less variation around the average and nodes more consistently reach a lower share as can be seen in column \bar{s}_x . Columns $s_{\bar{x}}$ and s_{s_x} show there is only very low variation of average shares and standard deviations among the different runs.

The maximum share of malicious nodes in a honest node's routing table is shown in column \max_x . It is the most interesting value because it decides whether the nodes that are off worst still have an honest majority in their routing tables increasing their chances to execute queries successfully. With $r = 4$, there are four scenarios where at least one node has half or even more than a half of the entries occupied by malicious nodes. These nodes will have difficulties executing storage operations. With higher replication factors, the maximum value stays below the 50% threshold in all scenarios. The same dropping of malicious nodes' shares can be seen in the average of maxima $\overline{\max_x}$ among the runs. Going from $r = 4$ to $r = 9$ reduces the maximum malicious node share by around 40% across the board. Increasing r to $r = 16$ provides a smaller improvement. The standard deviation of maximum shares among the runs s_{\max_x} also drops considerably as r is increased from $r = 4$ to $r = 16$ meaning for high replication factors the maximum share is pretty much identically low on all runs.

Table 6.12 shows the shares of malicious nodes on the top layer of routing table in the same test runs. This layer is especially interesting because it decides whether a node can successfully start disjoint routing paths. The results are very similar to the results over the whole routing table with one exception: the maximum share \max_x is considerably higher especially for the low replication factor $r = 4$. In scenarios

Table 6.12: Share of malicious nodes in top layer of routing tables

Scen.	m/\bar{n}	r	$\bar{\bar{x}}$	\bar{s}_x	$s_{\bar{x}}$	s_{s_x}	\max_x	$\overline{\max_x}$	s_{\max_x}
H100	0.0043	4	0.0045	0.0384	0.0012	0.0055	0.6667	0.3667	0.1054
M5		9	0.0045	0.0233	0.0010	0.0024	0.2500	0.1375	0.0395
		16	0.0041	0.0161	0.0012	0.0020	0.1333	0.0867	0.0322
H100	0.0085	4	0.0090	0.0539	0.0028	0.0079	0.6667	0.4000	0.1405
M10		9	0.0097	0.0343	0.0022	0.0041	0.2500	0.2000	0.0645
		16	0.0085	0.0231	0.0021	0.0028	0.1333	0.1267	0.0211
H200	0.0085	4	0.0094	0.0549	0.0019	0.0056	0.3333	0.3333	0.0000
M10		9	0.0085	0.0320	0.0018	0.0038	0.3750	0.2000	0.0874
		16	0.0086	0.0236	0.0015	0.0019	0.2000	0.1200	0.0422
H100	0.0168	4	0.0177	0.0763	0.0035	0.0081	1.0000	0.5667	0.2250
M20		9	0.0174	0.0456	0.0035	0.0051	0.3750	0.2500	0.0833
		16	0.0171	0.0333	0.0011	0.0025	0.2667	0.1733	0.0466
H200	0.0168	4	0.0179	0.0773	0.0018	0.0039	1.0000	0.6000	0.2108
M20		9	0.0156	0.0434	0.0026	0.0032	0.3750	0.3000	0.0645
		16	0.0171	0.0333	0.0011	0.0025	0.2667	0.1733	0.0466
H100	0.0250	4	0.0274	0.0958	0.0040	0.0075	1.0000	0.7667	0.1610
M30		9	0.0258	0.0558	0.0038	0.0043	0.3750	0.3125	0.0659
		16	0.0257	0.0396	0.0043	0.0036	0.2000	0.1867	0.0281
H200	0.0331	4	0.0356	0.1088	0.0027	0.0051	1.0000	0.7667	0.1610
M40		9	0.0322	0.0623	0.0036	0.0034	0.5000	0.3500	0.0791
		16	0.0337	0.0473	0.0016	0.0049	0.3333	0.2333	0.0471
H200	0.0489	4	0.0523	0.1303	0.0041	0.0057	1.0000	0.8333	0.1757
M60		9	0.0473	0.0745	0.0052	0.0043	0.5000	0.3875	0.0395
		16	0.0492	0.0565	0.0030	0.0045	0.3333	0.2600	0.0492

with at least 20 nodes, it goes up to 100% for $r = 4$ so there are nodes with their top layer completely filled with malicious nodes. They will not be able to execute any queries successfully. The average of the maxima $\overline{\max_x}$ is also considerably higher than 50% in all scenarios with at least 20 malicious nodes. This situation improves with $r = 9$ and $r = 16$. Then, the average of maximum node shares $\overline{\max_x}$ is always below 50%. However, with at least 40 malicious nodes the malicious node share also goes up to 50% with $r = 9$. With $r = 16$ the maximum is 33%.

The higher share on the top layer compared to the whole routing table is due to the fact that the top layer is easier to tamper with as it does not put tight constraints on the entries. If malicious nodes intercept the RtRequest on a lower layer, they might respond to it but do not end up in the routing table as they do not fulfil the tighter lower level constraint. On the top layer, they will be added if they are in the correct zone. For the same number of malicious nodes, getting a large share becomes harder if the size of the top layer grows with increasing r . Therefore, $r = 16$ and for a reduced number of malicious nodes $r = 9$ contain their influence and also keep the maximum share down at levels allowing successful sending of messages into the majority of zones.

Eclipsing of Joining Nodes

Since nodes request their own areas after joining, the nodes in other replication zones will notice this area is not correct and the node was eclipsed as discussed in section 5.5. Therefore, eclipsing of joining nodes is only possible if the bootstrapping service happens to supply only malicious nodes or nodes that have already been eclipsed.

This can happen if there are malicious nodes in all zones and the bootstrapping service picks them. The probability for this to happen depends on the number of zones equalling the replication factor r . In all of our 240 test runs this only happened one time in one run of the scenario with $r = 4$ and 60 malicious nodes being started after 200 honest nodes. It never happened in any run with $r = 9$ and $r = 16$. Considering the fact a node can always bootstrap again if it suspects it was eclipsed, eclipsing does not seem to be a big danger if the bootstrapper service itself does not return a higher than proportional share of malicious nodes.

6.3.5 Discussion

Using a self-stabilizing algorithm, our overlay is able to reliably maintain its primary Delaunay graph structure as nodes join and leave the network ungracefully. It also fills its routing tables reliably and maintains them under node churn. Thus, using the routing table, the storage can store and retrieve data on disjoint paths replicating data to the different zones.

We have shown the maintenance to be scalable. The total traffic grows linearly with the number of nodes in the network while the consumed bandwidth per node remains constant. Higher replication factors increase the traffic only by a constant factor. However, this factor depends on the square of the replication factor due to the redundant messages to fill one routing table entry. With high replication factors, the routing table maintenance makes up the vast majority of messages and traffic. The routing table maintenance mainly employs multi-hop unicast messages using them to look for entries. Therefore, we can expect multi-hop communication of unicast user messages to be scalable as well.

Finally, the overlay is also resilient to attackers controlling a certain share of malicious nodes if they are not part of the network when first starting it. Nodes temporarily destabilizing the primary overlay structure can be detected and punished. Eclipsing of nodes is very unlikely to happen, especially with higher replication factors. The redundant paths of the routing table maintenance successfully contain the influence of malicious nodes. Their average share on all routing tables is proportional to their share in the network, even if they had a much higher share at some point. Higher replication factors actually ensure there are no honest nodes with more than half of their routing tables being occupied by malicious nodes.

Thus, the overlay has the necessary properties for the storage to be reliable, scalable, and resilient as shown in the next section.

6.4 Storage Evaluation

This section presents our results evaluating the storage. First, we describe the scenarios the storage was tested with. Then, we show reliability and scalability results without the presence of malicious nodes. After describing the behaviour of malicious nodes, we also show how many of these nodes can be tolerated by the storage depending on total number of nodes and the chosen replication factor.

6.4.1 Storage Test Scenarios

We tested the storage with different numbers of nodes. Instead of slowly increasing node count by setting churn parameters accordingly, we used a ten minute setup phase at the beginning of each test scenario starting the desired number of nodes at random points within this phase. When this phase is finished, normal churn kicks in targeted at keeping the number of nodes in the network constant. We used test scenarios starting up 200, 400, 600, 800, and 1000 nodes in the range of World of Warcraft player numbers reported in [97]. To keep these numbers roughly constant, we used inter-arrival distributions with expected values of 25, 17, 11, 9, and 7 seconds, respectively. Again, we used replication factors $r = 4, 9, 16$ to test the storage with varying levels of redundancy.

When this phase is finished, there is a small delay of three minutes until all nodes create the initial state of the world and start the update process. We used an initial assumed maximum latency l_{max} of two seconds, leading to an update interval length of ten seconds with timeouts of four seconds for player retrieves and six seconds for area retrieves. When these ten seconds have passed, all player characters of nodes in the network will activate and player characters will start to generate actions. We assume a maximum interaction propagation speed of six meters per second and a maximum interaction range of 40 meters. Therefore, the radius of the update area is 100 m larger than the radius of the maintenance area of each node.

In addition to the update process, every node requests the area of interest around its player character once per update interval representing the way the storage is used to check the correct state in the world. We used an AOI radius of 100 m, which is probably even bigger than the AOI used in most current MMORPGs. However, this larger area allows a node to circumvent the disadvantage of the storage to return state only for the specific update times.

It can use the storage to retrieve all player actions within the requested large AOI. Using these actions, a node can perform the same update calculation as the storage to calculate the current state in a sub-area of the AOI based on the old state of a larger area and all actions of players in that area. This way, a node can check states also for intermediate times not covered by the storage. The AOI is always retrieved two seconds after the update process started retrieving the update area. This way, most update area retrievals should already have finished but nodes will not have completed retrieving player actions so the partial answers to the AOI queries should have the same world time.

To thoroughly test the storage, it would be desirable to have tests run over days and weeks. Unfortunately, this is infeasible because of the high amount of messages that will be transferred and have to be processed by the network simulator. In the biggest scenario with 1000 nodes and $r = 16$, only simulating one minute of the update process takes nearly six hours and consumes 20 GByte of RAM on our test server. This severely limited the amount of time we could simulate the storage.

Instead of simulating a few longer runs, we decided to test many very short runs. Once the update process started and shortcuts are set up, there will not be much variation in the results obtained on each update because the same queries will be issued. There are only a few factors causing variation. First, different message latencies might cause queries from zones being late that arrived in time in former updates which may lead to different query results. Second, nodes joining and leaving may cause data loss and changes in the layout of the network leading to some of the queries being handled by different nodes. Third, movement of player characters causes nodes to request the actions of different player characters. All of these changes only happen slowly over the expensive simulation time. Therefore, using short runs with completely different setups is more suitable to evaluate the storage in a variety of situations.

The update process runs for only one minute executing six updates. During the last two intervals, node churn is deactivated so the storage has the chance to repair any loss of data that occurred and we can check whether all data is consistent and available at the end. Finally, the system runs idle for one minute to make sure all late messages also arrived.

Stored Data

Our virtual world consists of objects making up the state of the world. Every object has a GUID identifier, a name, and a position. There are two types of objects in our world: player characters and non-player characters. An NPC only has movement and rotation vectors, current health, and AI state in addition to the basic fields. Therefore, an NPC has a data size of 158 bytes. We used a density of 1000 NPCs per km^2 resulting in a total of 44890 NPCs in our 44,89 km^2 world. The total data size of NPCs was 6,76 MBytes.

Storing player characters takes considerably more space. Player characters have a big inventory of items each of which might optionally be enhanced. Character progression through quests in the world also needs to be stored. Player progression was

picked at random with new player characters with small state consuming 374 bytes and the biggest player characters consuming 12,24 KBytes. We stored 5000 player characters matching the maximum number of node IDs in the key space totalling in an estimated 30,8 MByte amount of player character data. The actions generated by player characters contain data like GUID of the creating player character, time, and type of action resulting in a data size of only 30 bytes.

Modelling Player Behaviour

Player characters and non-player characters are not stored at static positions. They move through the virtual world. They are not randomly distributed in the virtual world and their movement is not random as well [97]. Instead, players typically move between locations performing specific activities while non-player characters typically move in the area of a home location waiting for player characters to attack. Therefore, we need a model to describe the locations of player characters and non-player characters so objects move around in the world more realistically.

Unfortunately, traces of real player movement and interaction are not publicly available since commercial providers usually do not disclose this information. There are studies of player mobility using in-world facilities to locate players like [97] studying World of Warcraft or [86] and [118] studying Second Life. However, the provided information is either too coarse-grained [97, 118], containing only the current world zone players are located in, or the recorded player traces are not available [86]. Therefore, we developed a simple model to describe player behaviour and movement. This model is based on findings in [86] and also inspired by our own observations.

We assume there are a number of points of interest where players go, for example to solve quests they have been given. We placed 100 of these points at random locations in the world. Initially, 50% of all players are placed at random locations within a 50 m radius around a randomly picked point of interest. The remaining 50% were placed at random positions in the world. Active player characters can choose to follow three types of activities: being idle and not performing any actions, moving to a point of interest, and interacting at the point of interest.

Whenever an active player character has no activity to follow, he picks one of the three at random with probability 10%, 50%, and 40%, respectively. If he picks idling, he will stay in the same location for 10 minutes and will not generate any actions. If he picks moving, he will pick one of the 10 nearest points of interest at random and move there at a speed of 6 m/s until he arrived within a 50 m radius

circle around this point. During this time he will generate a new action every 1 to 5 seconds representing the low interaction frequency that is necessary when just moving. After arriving at the point of interest, the player character picks a new activity. If he picks to stay there and interact with the point of interest, he will randomly move within the 50 m circle for five minutes before picking a new activity. During that time he generates a new action every 300 to 1000 ms, representing the high interaction frequency while fighting.

Of the 44890 NPCs, one third were placed around points of interest. The other two thirds were placed at random locations in the world. NPCs then randomly wander around within a 40 m radius around their home location at a speed of 4 m/s.

This model creates a more realistic scenario for data to move around the world. Players may aggregate at points of interest creating hotspots challenging for the storage.

6.4.2 Reliability

To evaluate the reliability of the storage, we tested 10 runs of each scenario described in section 6.4.1 ranging from 200 to 1000 nodes. Combining these scenarios with the replication factors $r = 4, 9, 16$ results in 150 test runs. We first analysed the data stored at the nodes at the end of each run, to find out whether the storage stores the data reliably without losing it or creating inconsistencies. Afterwards, we also analysed the results of the queries recorded while the storage was running to find out how reliably the storage was able to return requested data.

Stored Data Evaluation

According to our test scenario, the storage should store 5000 player characters – some of which are active while the majority is inactive – and 44890 NPCs totalling 49890 stored objects plus any actions of players that have been generated and stored. Table 6.13 shows the amount of stored objects in all scenarios averaged over the ten runs in the third column o . The total number of replicas is shown in column o_r allowing to calculate the effective replication factor r_{eff} which is higher than r due to the replicas of dynamic objects in areas overlapped by the maintenance areas of multiple neighbours. Column d_r contains the size of all stored replicas in MByte. Finally, the column NPC contains the number of stored NPCs while NPC_r contains

Table 6.13: Average numbers for data stored at the end of all test scenarios

n	r	o	o_r	r_{eff}	d_r	NPC	NPC_r	r_{eff}^{NPC}
200	4	52,814	387,533	7.34	192.39	44,890	354,863	7.91
400	4	55,631	403,813	7.26	198.40	44,890	359,099	8.00
600	4	58,550	415,162	7.09	203.74	44,890	357,955	7.97
800	4	61,466	426,432	6.94	209.07	44,890	356,755	7.95
1000	4	64,392	438,479	6.81	214.55	44,890	356,264	7.94
200	9	52,825	846,261	16.02	426.49	44,890	772,834	17.22
400	9	55,747	896,682	16.08	442.71	44,890	795,131	17.71
600	9	58,606	920,584	15.71	454.01	44,890	791,547	17.63
800	9	61,499	949,782	15.44	466.75	44,890	792,873	17.66
1000	9	64,183	975,538	15.20	479.12	44,890	792,606	17.66
200	16	53,714	1,406,164	26.18	738.09	44,782	1,275,372	28.48
400	16	55,655	1,564,039	28.10	780.22	44,890	1,385,291	30.86
600	16	58,617	1,640,387	27.98	807.79	44,886	1,410,712	31.43
800	16	61,525	1,684,521	27.38	828.63	44,890	1,405,178	31.30
1000	16	64,287	1,722,009	26.79	847.67	44,890	1,395,413	31.09

the number of NPC replicas, allowing to calculate the effective replication factor r_{eff}^{NPC} of dynamic objects in the world.

The number of stored objects o grows with the number of nodes independent of the replication factor. Although the objects in the world are the same in all scenarios, the number of generated and stored actions increases with the number of nodes. The size of stored replicas scales with the number of stored objects o and the replication factor r . The effective replication factor r_{eff} is nearly twice as high as r but decreases with increasing node numbers. Higher node numbers increase the number and relative share of stored actions that are not stored redundantly in overlapping maintenance areas. This can also be seen considering r_{eff}^{NPC} being close to twice as high as r throughout all scenarios. On average, all dynamic objects are stored at two nodes in each of the replication zones, adding additional redundancy to prevent data loss and tampering with data. In two scenarios with $r = 16$, the number of stored NPCs did not equal 44890 meaning the storage lost data.

Throughout all of our runs with $r = 4$ and $r = 9$, replicas for all objects existed and no object was missing any of the r replicas. This was different in the runs with $r = 16$ as shown in table 6.14. There, three of ten runs in the scenario with 200

Table 6.14: Number of lost objects with $r = 16$

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
200	1	0	0	0	938	143	0	0	0	0
400	0	0	0	0	0	0	0	0	0	0
600	0	0	0	42	0	0	0	0	0	0
800	0	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	0	0

Table 6.15: Number of objects missing replicas in at least one of $r = 16$ replication zones

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
200	25091	6158	21652	3342	7508	13693	30856	8600	13952	14094
400	212	0	0	743	1	0	0	553	521	1361
600	543	0	0	84	0	0	0	0	0	0
800	0	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	0	0

nodes and two of ten runs in the scenario with 600 nodes lost all replicas for objects that should be in the world.

Looking at the number of objects that should be in the world but are missing replicas in at least one of the 16 replication zones shown in table 6.15, all runs with 200 nodes, five out of ten runs with 400 nodes, and two out of ten runs with 600 nodes are missing replicas. With 800 or 1000 nodes, no replicas are missing, showing a clear trend that the storage is more reliable with more nodes in the network. However, there were no missing replicas with $r = 4$ and $r = 9$, even in the scenarios with 200 or 400 nodes. A higher replication factor leading to less reliability might seem counterintuitive. However, we used the same world so the same amount of stored data in all scenarios. In scenarios with fewer nodes, each node is responsible for storing more data. Further increasing the replication factor increases the amount of data each node is responsible for. Since each node periodically refreshes its data during the update process, the load on each might increase up to the point where its available bandwidth is too small to answer the queries of other nodes in time and to retrieve the state it needs to update its own area.

Table 6.16: Number of nodes with an incorrect state

n	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
200	196	195	199	200	201	197	194	199	197	198
400	398	0	348	400	385	0	397	366	317	403
600	418	0	458	548	0	444	399	132	244	466
800	0	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	0	0

This can also be seen in table 6.16 showing the number of nodes not containing the correct state for an object, either because the state it retrieved during the update was already incorrect or because it was not able to retrieve all actions of a player character in its update area, consequently not being able to update the state correctly. In the runs with 200 nodes, virtually all nodes do not contain the correct state. The actual number of nodes at the end of each run is different as the number of joining and leaving nodes during the run is different. In the runs with 400 nodes, all except two runs had nodes with incorrect state and of the remaining runs most had the vast majority of nodes contain an incorrect state. The runs without incorrect states (R2, R6) also did not have any missing replicas and no lost data before so these runs worked completely correct. With 600 nodes, all except two runs had incorrect state but in these runs the number of affected nodes varied. This improving trend continued since no node contained incorrect state with 800 and 1000 nodes. Again, no runs with $r = 4$ and $r = 9$ contained objects with incorrect state.

We conclude the storage to store its data reliably. Even a replication factor $r = 4$ is sufficient to not lose data up to network sizes of 1000 nodes and the corresponding churn to keep the number of nodes stable. One reason for this is probably that just one zone not losing data allows other zones to recover its data. So even if three of four zones have nodes joining and leaving losing data, they will be able to recover it on the next update. However, the storage stores data reliably only with enough available bandwidth. Since it loses data with 200 or 400 nodes with $r = 16$ having a symmetric bandwidth of 100 MBit/s, the bandwidth requirements are quite high. We will further analyse the bandwidth consumption in section 6.4.3.

Query Evaluation

Table 6.17 shows the average numbers of queries performed by the storage after the update process has started, leaving out any queries performed while the network was building up. The total number of queries increases with the number of nodes and with the replication factor. The vast majority of queries are the retrievals of player actions, increasing with node numbers and replication factor. The numbers of all other types of queries only depend on the number of nodes. Scenarios with higher node numbers have more churn causing more nodes to join and leave triggering the respective queries. Similarly, the number of times nodes request their update area during the update process and the number of times they request their area of interest merely depends on the number of nodes and the number of update cycles. The only type of retrieval increasing with the replication factor is the number of requested player actions. A higher replication factor increases the responsibility areas of nodes as fewer nodes cover the world area in each replication zone. More player characters will be in the area retrieved for the update so more player actions will be retrieved by each node.

Looking at the share of queries that returned the correct result, all scenarios with $r = 4$ and $r = 9$ returned the correct result. Only the same scenarios with 200, 400, and 600 nodes storing incorrect data also had queries with incorrect results as shown in table 6.18. With 200 nodes, between 18% and 43% of player action retrievals did not return the correct result, either because the retrieval itself was late in all zones or because the storing of player actions did not finish in time in enough zones. The incorrect player actions caused more than 60% of update area and area of interest retrievals to return incorrect data. Node join and departure caused retrievals with incorrect results but since only a few nodes failed and joined the share of incorrect retrievals fluctuates a lot. The share of player action retrievals with incorrect results decreased down to less than 1% as node numbers increase to 600. However, the number of incorrect AOI and update area retrievals does not decrease accordingly. Even only very few action retrievals returning incorrect results cause many updated states to be incorrect leading to state retrievals returning incorrect results.

The durations of the different query types are shown in table 6.19. In each run, we recorded the duration of each query and calculated the median instead of the average which would be skewed by the timeout defining a maximum duration for each query type. Afterwards we averaged the median over the runs of each scenario. Retrieving player actions was the fastest type of query with a median of less than 100 ms far lower than the four second timeout for this query type. The duration

Table 6.17: Average numbers of queries after update process start

n	r	Total	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
200	4	12,425	10,001	2	11	1,206	1,205
400	4	27,032	22,213	3	16	2,400	2,400
600	4	42,595	35,383	3	18	3,596	3,595
800	4	59,532	49,918	5	22	4,794	4,793
1000	4	77,445	65,421	7	32	5,993	5,992
200	9	22,708	20,287	2	9	1,205	1,205
400	9	48,881	44,063	3	16	2,400	2,400
600	9	76,246	69,034	3	19	3,595	3,595
800	9	105,955	96,339	5	24	4,794	4,793
1000	9	136,735	124,713	7	32	5,993	5,992
200	16	34,194	31,786	2	8	1,199	1,199
400	16	77,576	72,761	3	14	2,399	2,399
600	16	122,463	115,255	3	15	3,595	3,595
800	16	167,493	157,882	5	20	4,793	4,793
1000	16	213,836	201,814	7	26	5,994	5,994

Table 6.18: Share of incorrect queries in runs with $r = 16$

n	Run	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
200	R1	0.0184	0.0000	0.0000	0.6516	0.5477
200	R2	0.0223	0.0000	0.0000	0.6616	0.6540
200	R3	0.0280	0.0000	0.0000	0.6578	0.6337
200	R4	0.0306	0.0000	0.0000	0.6473	0.5840
200	R5	0.0234	0.0000	0.0000	0.6631	0.6225
200	R6	0.0385	0.0000	0.0000	0.6497	0.6239
200	R7	0.0433	0.5000	0.1000	0.6377	0.6351
200	R8	0.0298	0.0000	0.0000	0.6590	0.6504
200	R9	0.0278	0.0000	0.0000	0.6536	0.6192
200	R10	0.0387	0.5000	0.0000	0.6550	0.6232
400	R1	0.0027	0.0000	0.0000	0.5161	0.4017
400	R2	0.0000	0.0000	0.0000	0.0000	0.0000
400	R3	0.0006	0.0000	0.0000	0.3145	0.2042
400	R4	0.0056	0.0000	0.0000	0.5814	0.4769
400	R5	0.0024	0.0000	0.0000	0.4265	0.2958
400	R6	0.0000	0.0000	0.0000	0.0000	0.0000
400	R7	0.0024	0.0000	0.0000	0.4011	0.2741
400	R8	0.0041	0.0000	0.0000	0.3115	0.2031
400	R9	0.0021	0.0000	0.0000	0.2775	0.1818
400	R10	0.0011	0.0000	0.0000	0.4668	0.3470
600	R1	0.0012	0.0000	0.0000	0.1727	0.0920
600	R2	0.0000	0.0000	0.0000	0.0000	0.0000
600	R3	0.0010	0.0000	0.0000	0.3200	0.2096
600	R4	0.0011	0.0000	0.0000	0.3875	0.2487
600	R5	0.0000	0.0000	0.0000	0.0000	0.0000
600	R6	0.0008	0.0000	0.0000	0.2499	0.1577
600	R7	0.0004	0.0000	0.0000	0.2279	0.1333
600	R8	0.0004	0.0000	0.0000	0.0366	0.0183
600	R9	0.0004	0.0000	0.0000	0.1039	0.0532
600	R10	0.0013	0.0000	0.0000	0.2282	0.1563

is independent of the number of nodes but increases slightly with higher replication factors, due to the longer waiting for more than $r/2$ results from different zones to return.

All the other queries are area queries with durations ordered by the expected size of the requested area. The AOI and the area of failed neighbours are the smallest areas with fastest area queries. Both retrieving the update area as well as retrieving the own area when joining take considerably longer. Still, the median is far away from the six second timeout used for area queries.

Interestingly, the duration of AOI retrievals, retrieving an area with constant size, seems to increase slightly with the number of nodes for $r = 4$ and $r = 9$. This can be explained by more nodes leading to longer paths. For $r = 16$, the duration actually decreases with increasing node number which is a strong indication of bandwidth saturation in the smaller scenarios.

All other queries scale proportionally with the replication factor and show the trend of decreasing duration with increasing node numbers. This can be due to the sizes of the requested areas shrinking with increasing node numbers as the Voronoi cells of nodes determine the requested area size for all other types of area queries. A second effect might be the higher available bandwidths as other queries also consume less bandwidth. The strong effect bandwidth saturation has can be seen clearly in the scenarios with 200 and 400 nodes with $r = 16$ with all durations being considerably higher than in the other scenarios. We will analyse bandwidth consumption in more detail in section 6.4.3.

We conclude the storage is able to answer queries reliably as long as bandwidth requirements are met. Then, query durations are mostly shorter than timeouts of the query types. However, as replication factors grow and node sizes decrease, the available bandwidth becomes a limiting factor. This is especially true as most queries retrieve player actions and not being able to retrieve the action of only one player in time causes an incorrect state update. This incorrect state will be reported to other zones on the next update and may finally cause the state to be incorrect in the majority of zones.

6.4.3 Scalability

To find out how the storage scales with varying node numbers, we recorded the total traffic generated and the consumed bandwidth at each node. We also recorded the additional delay caused by up- and downstream capacity of the nodes being

Table 6.19: Average of query duration medians in ms

n	r	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
200	4	74.34	584.89	138.59	444.25	89.43
400	4	73.75	412.22	168.48	377.48	101.20
600	4	73.85	571.66	159.75	366.58	111.05
800	4	73.95	464.20	143.90	368.92	120.37
1000	4	74.18	482.52	134.53	374.88	129.62
200	9	108.36	824.91	210.90	1,488.63	149.09
400	9	97.77	580.74	155.27	1,028.70	112.65
600	9	93.62	473.34	164.72	856.37	118.80
800	9	91.81	482.71	131.65	788.64	127.90
1000	9	90.56	446.12	131.17	749.80	137.52
200	16	225.00	2,853.31	1,546.92	3,906.53	2,244.79
400	16	130.36	1,602.45	489.53	2,586.33	906.89
600	16	115.99	956.52	343.28	2,047.59	455.43
800	16	109.82	1,284.39	219.90	1,713.81	260.07
1000	16	106.18	644.33	241.10	1,548.78	202.97

saturated to find out whether limited bandwidth already had an influence on message latencies.

Furthermore, we wanted to know how evenly the load is distributed among the nodes. We recorded the sizes of responsibility areas, the amount of data stored at each node, the sizes of requested areas, and the data sizes of returned results. We also evaluated how many nodes contributed to the result of a query for each query type and how many hops messages of the different query types took. This information allows us to characterize the traffic in the network with more detail and to find potential bottlenecks

Total Traffic

The total amount of data transferred during the one minute the update algorithm runs is shown in figure 6.10. The traffic increases linearly with the number of nodes suggesting the storage to scale well with the number of nodes. Higher replication factors lead to a higher absolute numbers and a higher slope, requiring the individual nodes to handle more traffic.

However, the absolute numbers show the huge amounts of data to be transferred in only one minute. Compared to the overlay maintenance traffic shown in section 6.3.3 transferring around 12 GByte over eight hours at maximum, the storage transfers around 96 GByte within one minute with the same settings, equalling 45 TByte over eight hours. In one update interval, 16 GByte are transferred. Considering the total 848 MByte of stored replicas (see table 6.13) in that scenario, every piece of data is transferred nearly 20 times on each update. The large amount of transferred data consumes a considerable part of the bandwidth of nodes as we will show in the next section.

Bandwidth per Node

Figure 6.11 shows the average amount of data transferred upstream per second on each node. The consumed bandwidth actually decreases with rising node numbers, making the storage appear to be perfectly scalable. However, the bandwidth only decreases because the amount of data is largely independent of the number of nodes. With more nodes, only a few more replicas of the active player characters and their events have to be stored in addition. Therefore, with more nodes, each node is responsible for maintaining less data causing less traffic on each node. However,

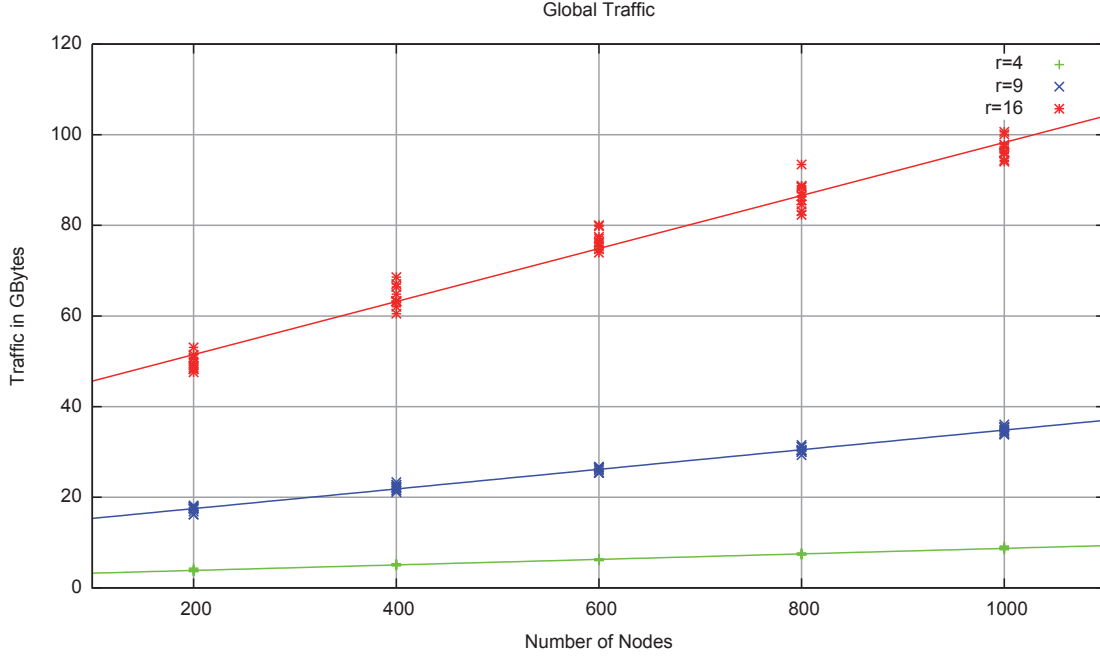


Figure 6.10: Global link layer traffic depending on the number of initially started nodes

the network size cannot be scaled indefinitely because at some point the world itself gets too small for many players and more objects have to be put into the world as well.

Table 6.20 characterizes the distribution of bandwidth consumptions among the nodes by averaging over all runs for each scenario. Considering the large deviations of minimum $\overline{\min}_x$ and maximum $\overline{\max}_x$ values from the average \bar{x} and median \tilde{x} and the standard deviation \bar{s}_x given, the distribution of load seems to be quite uneven with some nodes having to transfer much more data than others. In the extreme case with 200 nodes and $r = 16$, the 10 MByte/s maximum bandwidth is actually very close to the 100 MBit/s channel capacity limit. Since the update process creates bursty traffic pattern with most of the traffic occurring when nodes retrieve their update areas, the channel on these nodes is overloaded most of the time as we will show in the next section.

Channel Utilization

For every message sent and received on the link layer, we recorded the time the message was delayed because the upstream or downstream channel was busy. We calculated the average upstream and downstream delay for each node. To quantify

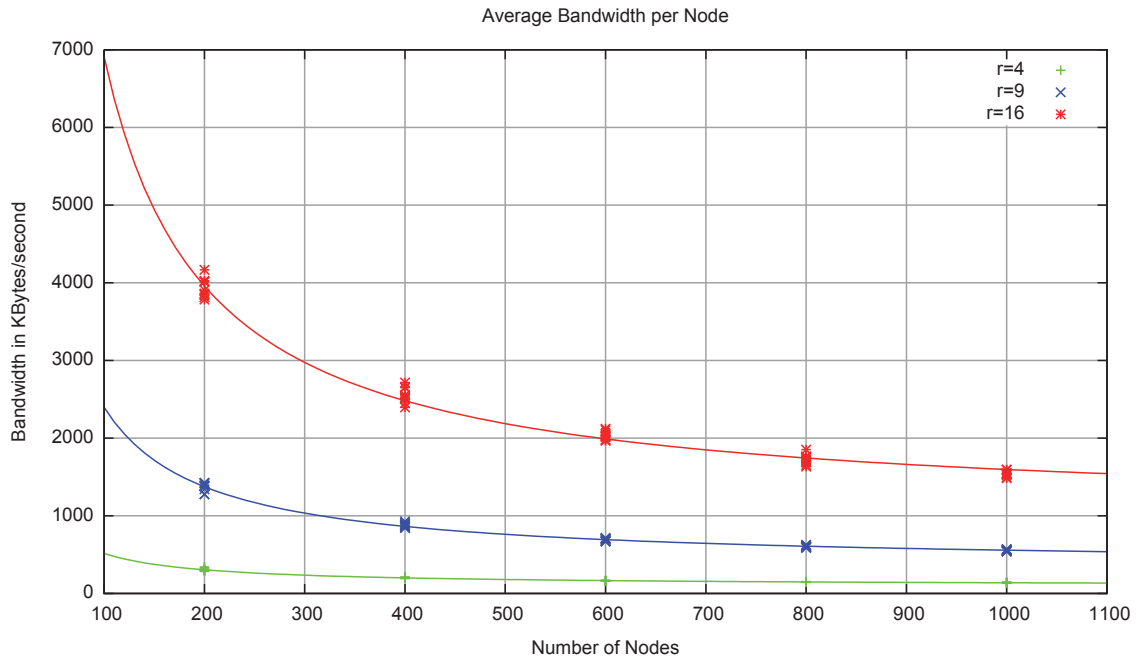


Figure 6.11: Average bandwidth per node depending on the number of initially started nodes

Table 6.20: Average values for bandwidth distribution characteristics

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min}_x$	$\overline{\max}_x$	\bar{s}_x
200	4	312,501	281,571	39,924	978,799	167,967
400	4	206,699	184,333	21,894	743,749	115,451
600	4	168,024	148,079	16,848	619,452	98,103
800	4	151,626	130,464	15,962	621,210	94,686
1000	4	142,510	120,135	13,686	623,968	94,200
200	9	1,402,574	1,282,548	207,646	4,326,873	746,766
400	9	900,363	822,925	97,234	2,978,757	484,893
600	9	706,211	639,285	63,156	2,520,191	379,140
800	9	619,602	562,754	68,448	2,313,356	339,014
1000	9	566,710	508,031	49,217	2,164,834	312,033
200	16	4,021,989	3,740,941	474,121	10,566,974	2,048,416
400	16	2,610,491	2,357,034	263,269	8,520,766	1,392,159
600	16	2,079,190	1,881,379	220,091	8,012,680	1,100,454
800	16	1,765,370	1,602,791	180,607	6,585,592	921,782
1000	16	1,576,419	1,442,091	157,521	5,729,352	816,822

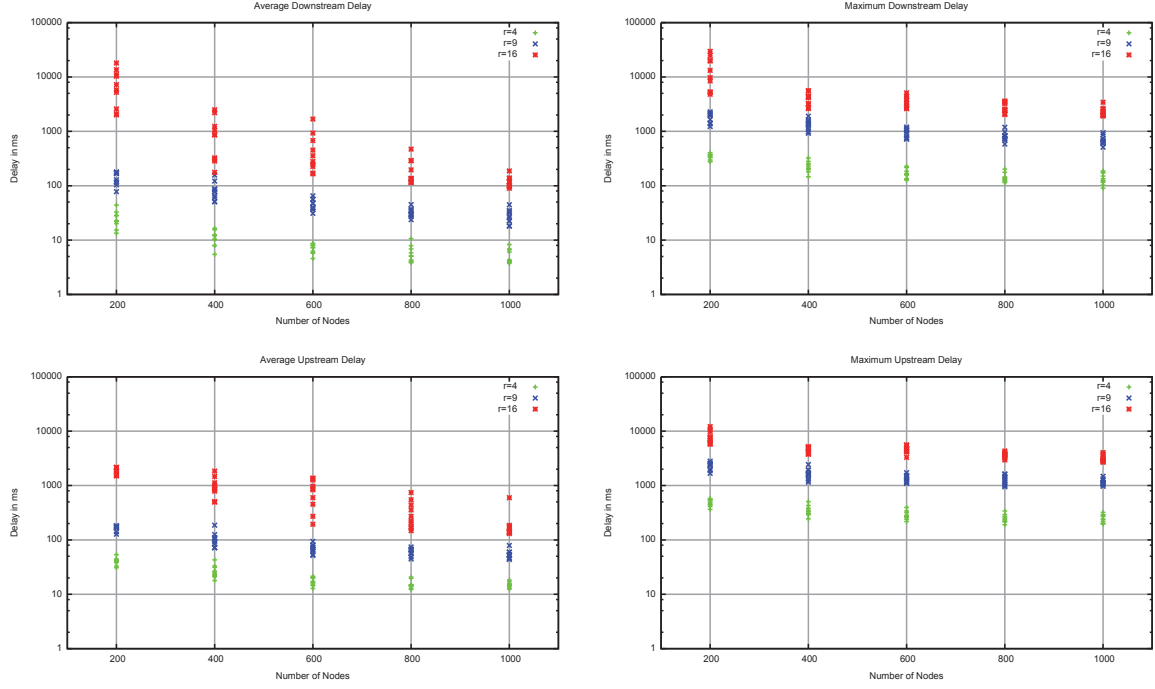


Figure 6.12: Average and maximum added delay due to channel saturation

the effect of high load on the ability of a nodes to process queries, we picked the node with the highest average and maximum upstream and downstream delay from each run. Figure 6.12 shows the results of this evaluation.

There is a general trend for upstream and downstream delay to decrease with increasing node numbers. This is consistent with the decreasing bandwidth consumption as update area sizes shrink and nodes experience less load. However, the delay decreases less as node number get larger than 400 in contrast to the bandwidth consumption that was decreasing at a similar rate all the way up to 1000 nodes. This shows the burstiness of traffic. The bandwidth consumption is also averaged over times with no message exchange while the delay is recorded only when messages are sent and received. Since all nodes retrieve data at the same time with generous timeouts to leave room for late messages, the bandwidth consumption will average out while the bursty update area retrieve phase still causes messages to be delayed.

The upstream delay generally tends to be slightly bigger than the downstream delay which is most likely due to the fact that nodes send out all the query messages to the replication zones at once on the upstream channel while the replies on the downstream channel will not arrive at the same time.

With $r = 4$, the average delay is always smaller than 100 ms and the maximum delay is always smaller than one second. Both values are well below the assumed maximum message latency of two seconds so bandwidth is high enough to process queries with minimum delay. With $r = 9$, the average delay is around 100 ms so most queries will be processed fast. However, the maximum delay increases to more than one second. In combination with a high Internet message latency, this value can actually cause some queries to not complete successfully reaching their timeout. In the scenario with 200 nodes and $r = 16$, the average message latency is around ten seconds meaning there is a node basically unable to retrieve any data or reply to any query in time. Going to 400 nodes, some runs still have a high average delay of one second while other stay below that mark. This correlates with the fact that some runs were already able to store data reliably in this scenario while others lost data. The maximum delay is still between one second and ten seconds, so some queries will not be processed in time. Increasing node numbers to 1000, the average delay with $r = 16$ shrinks to around 100 ms meaning most queries will be processed in time while the maximum delay is still between one second and ten seconds. With 1000 nodes, bandwidth is generally sufficient to only rarely cause queries to not be handled in time with replication factors $r = 9$ and $r = 16$.

Responsibility Areas and Stored Data

To find the cause for the high bandwidth consumption on some nodes inducing additional message delay, we analysed the sizes of responsibility areas and the amount of data nodes are responsible for. Table 6.21 contains the distribution characteristics for the amount of data stored on each node averaged over the ten test runs.

As expected, average \bar{x} and median \tilde{x} data sizes strongly depend on the replication factor and the number of nodes, increasing with higher replication factors and decreasing with higher node numbers. The large difference between minimum $\overline{\min_x}$ and maximum $\overline{\max_x}$ sizes and the large standard deviation \bar{s}_x show the distribution of data sizes to be quite uneven, similar to the bandwidth distribution. In the extreme case, nodes with the highest amount of data store 40 times as much data as nodes with the lowest amount of data. We analysed the correlation between bandwidth distributions and the amount of stored data by calculating the sample Pearson correlation coefficients between the distribution characteristics in table 6.22 using:

$$r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 * \sum_{i=1}^n (y_i - \bar{y})^2}}$$

Table 6.21: Average values of the distribution characteristics of the amount of data stored per node

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min}_x$	$\overline{\max}_x$	\bar{s}_x
200	4	1,001,903	896,530	89,077	3,491,244	586,639
400	4	519,579	454,540	41,002	2,038,621	324,674
600	4	356,020	305,940	28,681	1,572,228	238,941
800	4	273,875	223,161	18,281	1,287,423	190,866
1000	4	224,879	177,676	14,525	1,102,621	162,156
200	9	2,221,983	1,977,324	261,386	7,730,659	1,263,569
400	9	1,159,852	1,043,833	117,382	4,216,658	666,454
600	9	793,701	701,961	64,853	3,087,692	464,685
800	9	611,708	542,879	50,873	2,527,772	363,721
1000	9	502,412	440,936	41,016	2,084,350	302,606
200	16	3,845,811	3,433,743	431,467	13,552,313	2,180,240
400	16	2,044,399	1,829,569	172,586	7,316,606	1,161,363
600	16	1,412,377	1,270,325	128,280	6,188,916	800,310
800	16	1,086,139	974,240	107,865	4,251,297	607,026
1000	16	888,495	794,649	81,789	3,713,642	502,852

Table 6.22: Correlation coefficients of distribution characteristics

Distribution 1	Distribution 2	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min_x}$	$\overline{\max_x}$	\bar{s}_x
Bandwidth	Data	0.9070	0.9122	0.9289	0.8585	0.8981
Data	Voronoi	0.9994	0.9991	0.9917	0.9976	0.9994
Voronoi	Maintenance	0.9995	0.9997	0.9856	0.9934	0.9982
Data	Maintenance	0.9980	0.9979	0.9969	0.9880	0.9961
Update	Result Set	0.9820	0.9838	0.2743	0.9645	0.9778
Bandwidth	Result Set	0.9460	0.9443	0.1926	0.9350	0.9531

The correlation coefficients indicated a significant correlation between the amount of stored data and the bandwidth consumption. The only exception is the correlation coefficient of the maximum $\overline{\max_x}$. This is most likely due to the fact that there might be nodes storing a lot of inactive player character data that will not be requested during the update process.

If objects are evenly distributed, the amount of data stored should mainly depend on the size of a node's responsibility area. Therefore, once per update interval when the node queries its update area, we recorded the size of its Voronoi cell and of its maintenance area. Table 6.23 shows the characteristics of Voronoi cell sizes distributions averaged over the ten test runs.

As expected, average $\bar{\bar{x}}$ and median $\tilde{\bar{x}}$ Voronoi cell sizes show the same characteristics as the distribution of stored data. They strongly depend on the replication factor and the number of nodes, increasing with higher replication factors and decreasing with higher node numbers. The large difference between minimum $\overline{\min_x}$ and maximum $\overline{\max_x}$ sizes and the large standard deviation \bar{s}_x shows the same uneven distribution. The correlation between Voronoi cell size and the amount of stored data is shown in table 6.22, again indicating a significant correlation.

Since nodes store all dynamic objects inside their circular maintenance area and not only inside their Voronoi cell, we also recorded the sizes of maintenance areas yielding the distribution characteristics shown in table 6.24. The distribution characteristics are very similar to the characteristics of the Voronoi cell sizes, also showing a very strong correlation to Voronoi cell sizes as can be seen in table 6.22. However, the absolute difference shows maintenance areas to be more than twice as big as Voronoi cells on average. This is consistent with the additional numbers of replicas created due to overlapping areas as shown in table 6.13. Apparently, the shapes of Voronoi cells are not as compact as supposed resulting in this comparably

Table 6.23: Average values for Voronoi cell size distribution characteristics

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min}_x$	$\overline{\max}_x$	\bar{s}_x
200	4	893,382	796,776	107,613	2,925,142	486,386
400	4	448,825	406,616	63,643	1,506,821	238,767
600	4	299,595	266,798	49,244	1,137,822	159,662
800	4	224,679	201,638	44,617	858,612	115,858
1000	4	179,732	163,138	34,783	628,639	89,450
200	9	2,009,908	1,764,654	288,718	7,351,443	1,160,537
400	9	1,009,793	893,986	146,927	3,717,551	560,727
600	9	674,079	595,196	106,831	2,689,205	370,605
800	9	505,535	450,221	96,032	2,117,257	268,426
1000	9	404,397	360,888	83,422	1,651,821	209,560
200	16	3,542,585	3,085,236	450,883	13,884,378	2,114,902
400	16	1,795,159	1,558,933	224,575	6,597,142	1,034,184
600	16	1,198,367	1,060,174	179,571	5,388,986	680,129
800	16	898,629	796,718	159,476	3,447,660	488,084
1000	16	718,733	637,102	139,885	2,945,922	382,182

Table 6.24: Average values for maintenance area size distribution characteristics

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min}_x$	$\overline{\max}_x$	\bar{s}_x
200	4	1,958,678	1,762,719	274,100	6,771,109	1,054,042
400	4	964,931	860,826	154,547	3,490,365	511,146
600	4	633,954	565,456	89,815	2,515,736	335,348
800	4	470,399	422,121	75,402	2,450,055	245,418
1000	4	372,008	336,373	56,341	1,424,132	185,494
200	9	4,484,391	3,990,657	755,857	14,805,793	2,528,267
400	9	2,208,375	1,939,256	306,902	7,948,126	1,222,903
600	9	1,438,197	1,262,802	219,338	5,482,398	782,298
800	9	1,067,474	951,108	170,490	4,490,816	569,668
1000	9	847,111	753,480	139,950	3,676,084	444,652
200	16	8,351,765	7,140,488	1,135,380	34,686,912	5,204,762
400	16	4,009,264	3,455,413	437,957	14,816,166	2,330,788
600	16	2,652,790	2,326,981	358,452	11,599,003	1,536,663
800	16	1,944,689	1,697,430	277,241	8,853,076	1,094,088
1000	16	1,527,495	1,336,350	231,665	6,743,377	832,388

large increase in area size. On the other hand, the increased overlap created by the larger maintenance areas also has positive effects when it comes to the resilience against attackers.

The correlation between maintenance area sizes and the amount of stored data is also shown in table 6.22, suggesting the correlation to be as strong as the correlation of Voronoi cell sizes to the amount of stored data. The most likely reason for the correlation not becoming even stronger is the fact that the 4000 inactive out of 5000 player characters make up a significant share of the stored data. Inactive player characters are static data only stored once per zone based on the non-overlapping Voronoi cells. Therefore, the amount of stored data equally depends on the size of the maintenance area and the size of the Voronoi cell.

The bandwidth consumption does not directly depend on the amount of stored data or the sizes of maintenance areas. It depends on the amount of requested and transferred data. There is only an indirect connection as the requested data depends on the update areas determined by the maintenance areas. Therefore, we will analyse the correlation between bandwidth consumption and processed queries in the next section.

Query Result Sizes

The vast majority of transferred data can be accounted to the data retrieved during the update process. Messages to query or store data possibly routed over multiple hops are much smaller than the set of replicas returned in response to such a query. Therefore, we recorded the size of the returned result sets of the update area retrievals as well as the size of the retrieved areas to evaluate their correlation. Table 6.25 shows the distribution characteristics of the amount of data returned after retrieving the update area of a node. The amount of data transferred is considerably higher than the shown amount, as possibly redundant replicas are returned from each of the r replication zones.

Table 6.26 contains the same characteristics for the distribution of retrieved update area sizes. Since the update area is derived from the maintenance area by adding the maximum influence range to the radius, its sizes are generally bigger but the distribution characteristics show the same trend of increasing with replication factor and decreasing with node numbers. Furthermore, the distribution is just as uneven causing some nodes to retrieve much more data than others. The only notable difference is the larger relative increase in the minimum area size $\overline{\min}_x$ which is due

Table 6.25: Average values for update area retrieval data size distribution characteristics

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min}_x$	$\overline{\max}_x$	\bar{s}_x
200	4	457,507	422,019	66,593	1,335,537	217,050
400	4	285,373	262,272	40,262	920,329	137,999
600	4	224,183	203,553	27,272	752,130	113,375
800	4	194,860	174,856	23,271	714,515	102,647
1000	4	177,523	158,044	19,452	660,167	97,158
200	9	920,314	855,110	160,874	2,817,404	451,693
400	9	565,061	518,292	77,669	1,848,154	282,716
600	9	436,215	397,339	48,781	1,537,861	217,639
800	9	374,612	341,634	44,145	1,387,661	190,400
1000	9	336,991	305,741	36,154	1,216,179	172,041
200	16	1,474,925	1,358,009	0	4,907,005	763,911
400	16	935,665	849,367	111,770	3,164,804	476,111
600	16	728,036	662,009	86,953	2,763,084	370,125
800	16	614,299	559,467	67,917	2,287,304	310,333
1000	16	545,767	498,393	57,577	1,991,206	275,960

to the larger relative effect of adding the same constant maximum influence range to small areas.

The correlation of update area and result set size distribution characteristics is shown in table 6.22, suggesting very strong correlations between sizes of the retrieved areas and the contained data. The weak correlation of the minimum area and result set size are just a result of some queries in the scenario with $r = 16$ and 200 nodes not being able to complete successfully returning an empty result despite the large retrieved area. Looking at the correlation between result set sizes and bandwidth consumption in table 6.22, there is a strong correlation between the two. Only the minimum value appears to be uncorrelated but this is due to the same issue as before. We conclude the main reason for the uneven distribution lies in the uneven distribution of retrieved update areas. Since the latter depends on the maintenance areas which in turn depend on the Voronoi cells, the uneven distribution of Voronoi cell size is the major cause for the uneven bandwidth distribution. Other factors like uneven distribution of objects in the world at hotspots do not seem to have a major influence.

Table 6.26: Average values for update area size distribution characteristics

n	r	$\bar{\bar{x}}$	$\tilde{\bar{x}}$	$\overline{\min_x}$	$\overline{\max_x}$	\bar{s}_x
200	4	2,469,591	2,264,611	489,818	7,723,169	1,178,801
400	4	1,333,347	1,221,104	324,273	4,183,228	597,517
600	4	938,731	863,422	226,378	3,108,294	404,616
800	4	737,667	683,847	203,783	3,029,458	303,561
1000	4	613,523	573,378	171,610	1,877,254	235,924
200	9	5,239,094	4,730,160	1,092,301	16,195,553	2,725,470
400	9	2,747,941	2,464,274	534,053	8,976,225	1,359,649
600	9	1,880,583	1,692,529	415,368	6,343,249	889,351
800	9	1,453,524	1,328,233	347,374	5,269,075	659,845
1000	9	1,194,943	1,092,602	303,296	4,385,024	523,094
200	16	9,364,230	8,119,082	1,541,429	36,799,040	5,493,017
400	16	4,723,306	4,145,699	702,666	16,208,901	2,521,836
600	16	3,240,294	2,899,126	599,444	12,834,991	1,689,194
800	16	2,453,193	2,190,656	494,157	9,934,584	1,221,452
1000	16	1,982,691	1,777,544	432,879	7,693,245	941,907

The sizes of result sets and the bandwidth consumption decrease as node numbers increase because the world state is the same for all tested node numbers. To find out whether the number of messages a node has to receive to complete a query also scales with the network size, we recorded the number of messages necessary to complete a query and calculated the averages for the different query types. Since the number of messages will always increase with the replication factor, we also normalized it yielding the average number of messages from each replication zone. The results are shown in table 6.27.

In one zone, only one node stores the replicas of all actions of one player. From each zone, only one node will answer a request for player actions. After $r/2$ answers from all zones came in, the request can be completed. This means the number of messages is independent from the total number of nodes. It depends only on the number of zones.

The area of interest has constant size. The average number of nodes overlapping that area in a zone depends on the sizes of maintenance areas with bigger areas leading to fewer nodes overlapping the requested area. The maintenance areas are biggest with low node numbers and high replication factor and smallest with a high

Table 6.27: Number of messages per query result

n	r	Player Actions	Player Actions	Join Area	Join Area/ r	Neighbour Fail	Neighbour Fail/ r	Update Area	Update Area/ r	Area of Interest	Area of Interest/ r
200	4	2.00	0.50	10.04	2.51	5.67	2.83	21.29	5.32	3.03	0.76
400	4	2.00	0.50	11.53	2.88	5.00	2.50	23.98	6.00	3.68	0.92
600	4	2.00	0.50	11.05	2.76	4.99	2.49	25.58	6.40	4.21	1.05
800	4	2.00	0.50	10.78	2.70	5.40	2.70	26.96	6.74	4.71	1.18
1000	4	2.00	0.50	11.55	2.89	5.44	2.72	28.04	7.01	5.18	1.30
200	9	5.00	0.56	19.65	2.18	10.01	2.00	42.12	4.68	6.56	0.73
400	9	5.00	0.56	28.76	3.20	11.10	2.22	47.78	5.31	7.40	0.82
600	9	5.00	0.56	27.36	3.04	11.33	2.27	51.05	5.67	8.19	0.91
800	9	5.00	0.56	27.94	3.10	11.64	2.33	53.64	5.96	8.94	0.99
1000	9	5.00	0.56	27.21	3.02	12.52	2.50	55.64	6.18	9.59	1.07
200	16	8.01	0.50	37.91	2.37	23.06	2.88	63.67	3.98	10.30	0.64
400	16	8.01	0.50	44.89	2.81	25.29	3.16	76.74	4.80	11.34	0.71
600	16	8.00	0.50	48.60	3.04	22.26	2.78	82.62	5.16	12.05	0.75
800	16	8.00	0.50	54.12	3.38	23.35	2.92	85.14	5.32	12.48	0.78
1000	16	8.00	0.50	48.71	3.04	23.28	2.91	88.07	5.50	13.16	0.82

number of nodes and a low replication factor. Thus, the scenario with 200 nodes and $r = 16$ features the smallest average number of messages per zone while 1000 nodes and $r = 4$ features the highest for retrieving the area of interest.

The same trend of increasing message numbers can be seen in the columns for join area and neighbour fail retrievals, despite the higher variation due to the low number of node joins and departures. The sizes of the requested areas are dynamic and depend on the area sizes of the involved nodes. Although the area sizes are bigger with higher replication factors, the number of messages per zone is about the same for the same number of nodes but different replication factors. As the requested area sizes vary, so do the areas of the nodes they overlap and the number of overlapped areas stays the same.

However, for the retrieval of the update areas, we see the message numbers per zone increase as replication factors and maintenance area sizes decrease. This is again an effect of the requested update area being the maintenance area increased by the maximum influence range. With smaller maintenance areas, the requested update areas become larger in relation and will overlap more maintenance areas at the destination.

This actually suggests that there is a second limit to the scalability in addition to the world and its content being too small for increasing player numbers. As player numbers go towards infinity, the maintenance area sizes will converge towards zero. However, the requested update area sizes will always have a constant minimum size with the maximum influence range as radius. Therefore, the number of maintenance areas overlapped by the requested update area and thus the number of messages per query will also converge towards infinity. At the same time, most nodes will probably just send back empty answers as the area they are responsible for does not contain any data and the total amount of transferred data still only depends on the size of the requested area. However, we expect this effect to become a limit only for much higher node numbers. In our scenarios, the average number of messages from each zone lies between four and seven which is still reasonable value. If the number of nodes really becomes too high in relation to the replication factor, the replication factor can always be increased.

Path Lengths

Returning results to area queries makes up most of the data transferred by the storage. As query response messages are always transferred directly, they will generally

Table 6.28: Average hop counts per query type

n	r	Store	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
200	4	0.7502	1.1858	1.8248	1.7633	1.7636	1.4817
400	4	0.7501	1.2697	1.8616	1.8320	1.8178	1.6639
600	4	0.7501	1.3233	1.8831	1.8022	1.8474	1.7729
800	4	0.7501	1.3631	1.8919	1.7851	1.8722	1.8560
1000	4	0.7501	1.3971	1.8995	1.7726	1.8936	1.9270
200	9	0.8890	1.1097	1.7561	1.7073	1.8653	1.3190
400	9	0.8890	1.1768	1.8156	1.7640	1.9185	1.4576
600	9	0.8890	1.2167	1.8416	1.7625	1.9401	1.5440
800	9	0.8891	1.2456	1.8570	1.7714	1.9598	1.6099
1000	9	0.8890	1.2694	1.8655	1.7867	1.9748	1.6597
200	16	0.9380	1.0703	1.7106	1.7559	1.8943	1.2432
400	16	0.9376	1.1298	1.7865	1.7683	1.9446	1.3595
600	16	0.9376	1.1670	1.8170	1.7949	1.9733	1.4403
800	16	0.9377	1.1954	1.8368	1.7639	1.9849	1.5029
1000	16	0.9376	1.2163	1.8472	1.7670	1.9947	1.5499

need only one hop. Queries might still be routed over multiple hops. Therefore we evaluated the average path lengths of the different query types in table 6.28.

For all types of messages, the average hop count is smaller than two. For messages to store player actions the average path length is actually smaller than one, because the replica in the storing node's zone is stored at the node itself. Furthermore, every player always sends his actions to the same positions so there is not much change in the paths.

The set of retrieved player actions changes depending on the players in a node's update area so the path length should be logarithmic with the number of nodes. The path length actually increases with the number of nodes, but only slightly. Apparently, the set of players in the update area does not change fast so shortcuts will be used to send the following queries to retrieve the actions from the same player so the average path length gets closer to one.

All area retrieve messages have a longer path length than action retrieval messages as they first have to reach a point in the requested area and create a multicast tree covering that area afterwards. For neighbour fail, join area, and update area retrievals, the retrieved area is related to the node's own area and the routing table will cause the retrieve message to reach the target area in one hop only. The number of hops in the multicast tree depends on the size of the requested area, so it slightly increases with the increasing area sizes of neighbour fail, join area, and update area retrievals.

The fact that the average path length for update area retrievals is still below two might surprise. Requesting an area larger than the average Voronoi cell size should on average create a multicast tree with minimum height one as at least two nodes should be covered by the requested area. However, the request message to the node's own replication zone does not need the one hop to get into other zones. The node will start the multicast directly itself so saving this one hop brings the average down below two. The effect of this one zone also diminishes with increasing replication factor.

Finally, the average path lengths for retrieving the area of interest is mostly still below the values for update area retrievals. Although the retrieved AOI changes with the current player position and retrieve messages should be routed to the destination zones over multiple hops, it does not change fast enough so shortcuts will keep the length of the first part of the path down to one hop most of the time. The height of the multicast tree is lower than the height for update area retrievals since the AOI is smaller, creating the smaller average values. One exception is the scenario with 1000 nodes and $r = 4$. There, Voronoi cells got so small that the constant AOI size together with the longer paths to the multicast start cause a slightly higher path length in that case.

In general, the average path lengths stays very short as node numbers increase. Even if the maximum values are bigger, upon the second retrieval of the same data, shortcuts will keep the path lengths small. Therefore, increased message latency due to multi-hop routing should not have a negative impact on the ability of the storage to process queries and update its data as the network size increases.

6.4.4 Resilience

To determine the resilience of the storage, we had to find out how many malicious nodes the storage can tolerate before they were able to successfully tamper with

data. The test scenarios and the behaviour of malicious nodes will be described in the following two sections. Afterwards, we first analyse the data at the end of each test run looking for data that was successfully tampered with. Second, we also evaluate the influence of malicious nodes on the different types of queries to find out which types are most vulnerable.

Scenario Setup

From our storage test scenarios described in section 6.4.1, we picked the largest scenario with 1000 nodes and an inter-arrival distribution mean value of 7 seconds. Again, we used $r = 4, 9, 16$ and performed ten test runs each varying the number of malicious nodes trying to find a scenario where data starts to get tampered. From the thousand nodes, we first started the honest nodes and the malicious nodes last. As we have shown, starting the malicious nodes earlier would not make a difference as they will always only have a proportional influence. In the different scenarios, we only varied the number of malicious nodes – the remaining setting deciding on node IDs, times of join and departure, and on message latencies were identical. Therefore, if there are two runs, one of a scenario with 20 malicious nodes and another one with 21 malicious nodes, they will run identical except for the fact that node number 980 will additionally act maliciously on the second run. More precisely, the times of node join, node departure, and node IDs are the same. Upon the first node sending a different message, latencies for messages will be different. However, it is still easier to compare runs of different scenarios to find the cause for data tampering this way.

Malicious Node Behaviour

The goal of a malicious node is to modify the stored data so it does not represent the correct state of the virtual world. We decided to give the attacker as much power as theoretically possible so we can determine how resilient the storage really is.

A malicious node will act very aggressively, tampering with every type of query if it gets a chance to. There is no procedure in place to detect and punish tampering nodes so trying to tamper with everything will give malicious nodes the biggest influence. Furthermore, where coordination of malicious nodes is necessary to tamper successfully, we assume the attacker can coordinate tampering perfectly in zero time

using all the information available to his malicious nodes. Malicious nodes also continue to try to get an increased share in honest nodes' routing tables as described in section 6.3.4.

When a malicious node receives an area query and the multicast has already started, it can only tamper with data in its responsibility area. If the query is requesting dynamic data, the malicious node will not tamper with data overlapping the maintenance areas of honest neighbours that will also receive the query as shown in figure 5.6. Doing this would create conflicting versions of the same object and the receiver will discard the result from this replication zone. Therefore, the malicious node first calculates the non-overlapping area where it can actually tamper with data. If static data is requested, data in its whole Voronoi cell can be modified.

A malicious node can only tamper successfully if it obtains at least a relative majority for its tampered replicas. When the first malicious node receives a query, it can already predict which malicious nodes in other replication zones will receive the same query and it also knows which areas they can tamper with without creating conflicting versions. Then it calculates the sub-area of the requested area where most of its tampering areas overlap. It will tamper with all data in this area and return the result. When the malicious nodes in the other areas receive the query, they will tamper with the same data and return it. Finally, the attacker will have a number of votes for the tampered state equalling the number of replication zones where he had overlapping areas of tampering.

If a malicious node receives a query but its tampering area does not overlap the tampering areas of other malicious nodes, it will still tamper with data to prevent a correct result from being sent back. It is still better for a malicious node to have another incorrect answer in the voting set at the receiver because it increases its chances to get a majority for a tampered result.

Figure 6.13 shows how the responsibility areas of malicious nodes mapped to the world area overlap the requested area A . Node a joins and requests its own area A . Nodes m_1, \dots, m_4 are malicious nodes in different replication zones. Their responsibility areas M_1, \dots, M_4 overlap the requested area A in their respective zone. M_1 , M_2 , and M_3 all overlap A in a small common area T . By tampering only with data in this area m_1 , m_2 , and m_3 can obtain three votes for a tampered state. Node m_4 does not overlap T , but also tampers with data in its area to reduce the number of correct votes.

If a malicious node receives an area query where the multicast has not started yet, it can claim responsibility for the requested area by reporting a wrong neighbourhood.

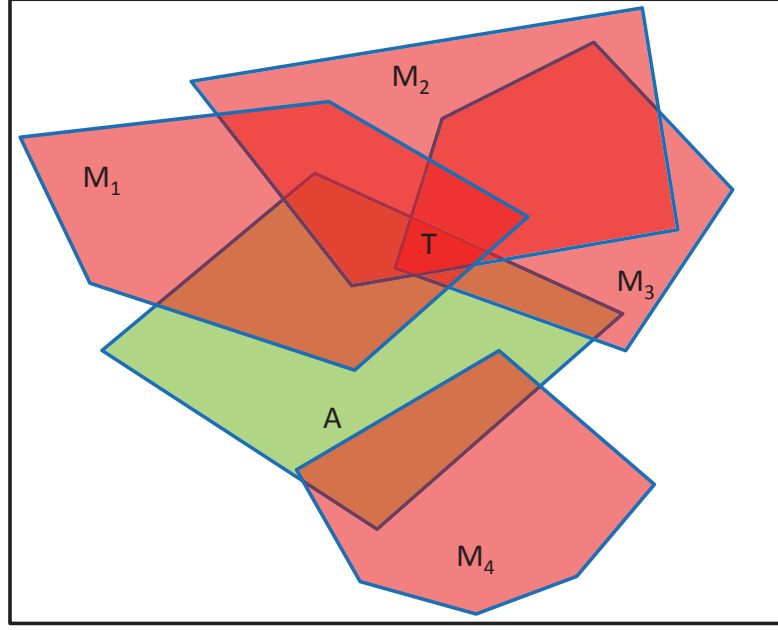


Figure 6.13: Malicious nodes overlapping a requested area tamper with data in area T to obtain three votes

Therefore, it will reply with a tampered result matching the objects the attacker could tamper with in other zones. If there are no other zones where the attacker controls malicious nodes also receiving the query, the node will make up tampered data in the requested area anyway.

In addition to tampering with area queries, malicious nodes will also tamper with retrievals of stored player actions and with the storing of player actions themselves. They will try to prevent honest nodes from calculating correct updates of the world state by dropping actions of honest players. Whenever a malicious node receives a request for player actions, it claims being responsible for storing these actions and replies with an empty result, pretending the player did not perform any actions in the latest interval. When a malicious node receives actions of players to be stored, it neither stores nor forwards these actions to the responsible nodes. This way the actions of players will not be stored and even honest might not be able to answer queries for player actions correctly. Although this kind of tampering does not allow to modify the state in any way like tampering with area queries does, we do not differentiate between the two types as still malicious nodes were able cause incorrect state to be created.

When an honest node retrieved a result containing tampered data and updates the objects according to the retrieved result, the tampering will spread out to all objects

within the maximum influence range of any retrieved tampered object. Since the last state of the object was tampered, any other object this object could have influenced up to the current update time must be considered as being tampered. The same is true for any missing actions of a player character. All objects close enough to this player character will be considered tampered. Spreading tampering this way is a very conservative worst-case assumption as in practice, most objects will not be affected this way. However, as this case can theoretically arise and we want to estimate the worst-case resilience, we modelled tampering of state spreading out this way.

Stored Data Evaluation

Table 6.29 shows the influence a varying number of malicious nodes had on the stored data in the ten test runs with $r = 9$. The first line of each scenario with m malicious nodes shows the number of honest nodes storing replicas that have been tampered with. The second line shows the number of objects that have been tampered successfully because the majority of replicas is tampered.

While 15 malicious nodes cannot cause any honest node to store tampered replicas in any run, in run R3 with 18 and 19 nodes there is one honest node storing tampered replicas. The same run is also the first where some objects were tampered with successfully, while the other runs still do not contain tampered data. Apparently, this run has some properties, probably the positions of nodes, giving the malicious node the opportunity to tamper. As expected, data continues to be tampered in this run as the number of malicious nodes is increased. With 25 malicious nodes, the run R10 is the second run where honest nodes store tampered replicas and some objects are tampered. With 30 malicious nodes, half of the runs contain tampered data. With increasing number of malicious nodes, even more runs have tampered data.

It is interesting to note that in the same run, increasing the number of honest nodes containing tampered replicas does not necessarily mean more objects are tampered successfully. Although more nodes store a tampered replica, tampered replicas might still be in the minority. A possible reason is that the replicas got tampered just recently and the majority being tampered will only happen at the next update. However, it is also possible some honest nodes will continue to retrieve tampered data because they have multiple malicious nodes in their top routing table layer. If tampered replicas are still in the minority, nodes responsible for the same data in other replication zones might still be able to obtain the majority for the correct

Table 6.29: Honest nodes with tampered data and data with tampered majority for $r = 9$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
15	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
18	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
19	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
20	0	0	38	0	0	0	0	0	0	0
	0	0	242	0	0	0	0	0	0	0
25	0	0	38	0	0	0	0	0	0	99
	0	0	235	0	0	0	0	0	0	663
30	37	0	39	0	0	0	87	0	0	98
	36	0	241	0	0	0	510	0	0	703
35	46	0	75	0	136	0	85	0	0	100
	106	0	152	0	928	0	118	0	0	427
40	157	99	175	77	141	2	248	92	0	216
	1354	642	1695	266	1039	0	1248	459	0	1112
50	198	366	196	243	244	166	310	287	158	369
	1375	2065	2055	1319	1482	802	1977	2560	847	2094
75	711	820	686	761	623	595	712	755	694	821
	5694	7232	4617	5568	4570	4567	6539	7654	5513	7114

state. In the presence of malicious nodes, the storage cannot guarantee eventual consistency even among the honest nodes only. However, in the end the decisive factor is that the majority of replicas are not tampered with and this majority can be retrieved by requesting nodes.

With regards to the level of resilience, with $r = 9$, the storage can only tolerate up to 19 out of 1000 nodes being malicious equalling a 1.9% share of malicious nodes before data gets tampered. First signs of tampering already appear at 1.8%. Considering we tested only ten runs over a short period of time, the actual share of malicious nodes the storage can withstand in all scenarios might still be lower.

Comparing this to the results with $r = 4$ shown in table 6.30, only two out of 1000 nodes being malicious could be tolerated in all ten runs, equalling a share of 0.2%. Although most runs could tolerate up to six malicious nodes, run R6 already created

tampered data with only three malicious nodes. Trying to find the cause for this low resilience, we found reaching a constellation to tamper data is quite easy with $r = 4$. It was not necessary to really have overlapping areas with malicious nodes in multiple zones. One node just had a malicious node in its top routing table layer. Requesting its update area, the node got one reply from a zone where a malicious node had a maintenance area with parts not overlapping its neighbours' maintenance areas. The malicious node in the top layer also created a matching tampered vote. The results from the other two zones were correct but since both votes were equal, the tampered one was picked randomly. Considering that churn might also lead to zones not being able to reply with correct results, it is clear that a replication factor of $r = 4$ is not sufficient to provide a useful level of resilience. It is also interesting to see that the timing of messages also has a big influence on tampering. In run R1, tampering was possible with eight malicious nodes but not with nine or ten nodes. Apparently, the different behaviour of the additional malicious nodes changed the timing of messages in a way that tampering was no longer possible, although there were more malicious nodes.

With $r = 16$, up to 38 malicious nodes could be tolerated in all ten runs as shown in table 6.31. With 39 malicious nodes, the first run contained tampered data. This means a share of 3.8% malicious nodes could be tolerated. With 60 malicious nodes, four out of ten runs still did not contain tampered data while with 75 malicious nodes, data was tampered in all runs. Compared to $r = 9$, twice the number of malicious nodes could be tolerated with a little less than twice the replication factors. However, this increase in resilience comes at a high bandwidth cost and the level of resilience is still quite low. Furthermore, with more and longer test scenarios, the final level might still be lower.

Query Evaluation

Table 6.32 contains the share of queries malicious nodes tried to tamper with. For area retrievals, they were able to tamper the replica set from at least one replication zone. For player action retrievals, malicious nodes either dropped action replicas returned from at least one zone, they intercepted the storing of the actions, or storing of a replica was delayed and so it could not be retrieved.

Since player actions make up the vast majority of retrievals as shown in table 6.17, the share of queries malicious nodes try to tamper with of all queries mainly depends on the share of player action retrievals. As shown in table 6.32 for $r = 9$, the total share grows from 13% with 15 malicious nodes to 45% with 75 malicious nodes. At

Table 6.30: Honest nodes with tampered data and data with tampered majority for $r = 4$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
1	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	27	0	0	0	0
	0	0	0	0	0	66	0	0	0	0
4	0	0	0	0	0	28	0	0	0	0
	0	0	0	0	0	66	0	0	0	0
5	0	0	0	0	0	27	0	0	0	0
	0	0	0	0	0	66	0	0	0	0
6	0	0	0	0	0	74	0	0	0	0
	0	0	0	0	0	598	0	0	0	0
7	0	0	0	3	0	77	0	55	0	0
	0	0	0	0	0	708	0	775	0	0
8	14	1	0	0	0	76	0	69	0	106
	110	0	0	0	0	651	0	1274	0	1677
9	0	24	63	0	0	75	47	72	0	105
	0	326	1225	0	0	676	626	1346	0	1434
10	0	0	68	49	51	78	105	134	48	105
	0	0	1200	380	684	670	1437	2334	848	1426

Table 6.31: Honest nodes with tampered data and data with tampered majority for $r = 16$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
35	0	0	0	0	0	0	4	0	0	1
	0	0	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
39	0	0	92	0	0	0	0	0	0	0
	0	0	189	0	0	0	0	0	0	0
40	0	0	92	0	0	0	0	0	0	0
	0	0	233	0	0	0	0	0	0	0
45	0	0	89	0	0	0	0	56	0	0
	0	0	164	0	0	0	0	94	0	0
49	0	0	80	2	0	89	0	86	0	0
	0	0	60	0	0	213	0	415	0	0
60	1	0	103	151	51	140	3	165	87	2
	0	0	545	438	47	891	0	795	283	0
75	61	236	213	282	94	252	249	348	334	256
	187	834	512	1164	425	975	418	2085	1791	1112
100	565	585	697	675	491	674	637	717	644	543
	3480	2844	3979	3401	2879	2417	3999	3785	3393	2314

Table 6.32: Share of queries malicious nodes tried to tamper with for $r = 9$

m	Total	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
15	0.1302	0.1080	0.2857	0.1344	0.5697	0.1753
18	0.1497	0.1252	0.2794	0.1734	0.6296	0.2064
19	0.1564	0.1311	0.3595	0.1837	0.6491	0.2171
20	0.1622	0.1361	0.3235	0.1548	0.6707	0.2251
25	0.1992	0.1708	0.4203	0.1893	0.7471	0.2787
30	0.2319	0.2020	0.4853	0.2809	0.8033	0.3239
35	0.2622	0.2303	0.4853	0.3105	0.8589	0.3751
40	0.2910	0.2585	0.6176	0.3247	0.8945	0.4160
50	0.3416	0.3084	0.6912	0.4271	0.9388	0.4942
75	0.4494	0.4162	0.8000	0.5889	0.9868	0.6801

the same time, the share of update area retrievals malicious nodes tried to tamper with increases from 57% to 99%. This much bigger share with 15 malicious nodes shows that a lot more nodes are involved in retrieving an update area causing even a small share of malicious nodes to be involved in nearly all update area retrievals. The shares of the other area retrievals are smaller, depending on the sizes of the requested areas. With 75 malicious nodes, data got tampered successfully so even honest nodes return tampered data and nearly all update area queries are tampered. Since player actions must be tampered with directly instead of tampering just spreading out, only malicious nodes involved in the query can actually try to tamper and the share of player action queries is lower at 42%

With $r = 4$ and less malicious nodes, the shares are much lower as shown in table 6.33, only increasing above 10% for update area retrievals as data got tampered successfully. On the other hand, with $r = 16$ shown in table 6.34 the share was much higher with 96% of update area queries being influenced by only 35 malicious nodes. However, due to the higher replication factor the storage is able to tolerate these high shares without data getting tampered. Apparently, the share of queries malicious nodes try to tamper with mainly depends on the share of malicious nodes.

Table 6.33: Share of queries malicious nodes tried to tamper with for $r = 4$

m	Total	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
1	0.0082	0.0063	0.0000	0.0000	0.0301	0.0084
2	0.0166	0.0126	0.1449	0.0938	0.0616	0.0174
3	0.0253	0.0194	0.1449	0.0935	0.0923	0.0278
4	0.0328	0.0246	0.1932	0.0935	0.1218	0.0393
5	0.0407	0.0308	0.1449	0.0719	0.1509	0.0454
6	0.0470	0.0357	0.1449	0.0762	0.1739	0.0515
7	0.0553	0.0423	0.1449	0.0752	0.2017	0.0608
8	0.0619	0.0471	0.1449	0.0752	0.2258	0.0708
9	0.0705	0.0546	0.1449	0.0815	0.2477	0.0797
10	0.0790	0.0609	0.1812	0.1333	0.2768	0.0934

Table 6.34: Share of queries malicious nodes tried to tamper with for $r = 16$

m	Total	Player Actions	Join Area	Neighbour Fail	Update Area	Area of Interest
35	0.3309	0.3089	0.8551	0.5412	0.9574	0.5027
37	0.3466	0.3247	0.7887	0.5000	0.9638	0.5267
38	0.3521	0.3304	0.8696	0.5292	0.9638	0.5321
39	0.3857	0.3651	0.8571	0.5018	0.9660	0.5640
40	0.3925	0.3722	0.8451	0.5075	0.9676	0.5699
45	0.4234	0.4035	0.8986	0.5568	0.9795	0.6102
49	0.4465	0.4271	0.9130	0.5625	0.9842	0.6389
60	0.5024	0.4842	0.9565	0.6176	0.9944	0.7083
75	0.5664	0.5504	0.9855	0.7000	0.9983	0.7720
100	0.6482	0.6345	0.9718	0.8115	0.9997	0.8673

The queries malicious nodes could actually tamper with are shown in table 6.35 for $r = 9$. The first line for each scenario contains the number of tampered player action retrievals while the second line contains the number of tampered update area retrievals. As expected, for both types the number of tampered queries increases as the number of malicious nodes grows. In runs where data was successfully tampered as shown in table 6.29, there are both tampered player action queries and update area queries. Conversely, there are runs with tampered player action queries and update area queries that did not lead to data being tampered, e.g. the run R3 with 18 and 19 malicious nodes. In these runs, there is probably one node retrieving a tampered update area five times without causing the majority of replicas being tampered. There are also runs where player action retrievals were tampered causing nodes to calculate incorrect updates, while not leading to tampered update area retrievals. Apparently, the majority voting suppressed these incorrect replicas.

Probably most interesting is the fact that tampered update areas never appear without tampered player action retrievals in contrast to the opposite direction. The most likely explanation is that tampered player actions are actually the cause for state getting tampered. Only the combination of some actions not being retrieved successfully and some update area retrievals being tampered allows final successful tampering. In these cases, cheaters can only tamper the state by dropping actions of players. They cannot modify the state in any way they desire.

Only one run is an exception: the run R10 with 15 malicious nodes actually had one tampered update area retrieval but no tampered player action retrieval. However, as this is only one retrieval, it probably suffered from late messages returning the results of honest nodes while only malicious nodes answered before the timeout. On the next queries, the node was able to retrieve its update area correctly.

Looking at the same data obtained with $r = 4$ in table 6.36, there are actually runs like R6 with tampered update area queries without tampered player action queries. The reason for this is scenario we sketched before with one malicious node in the top routing table layer and another zone containing a tampered object is sufficient to tamper successfully, reinforcing our thesis that $r = 4$ is not sufficient to provide a useful level of resilience.

On the other hand, the results obtained with $r = 16$ in table 6.37 are again very similar to the results with $r = 9$. There was no tampered update area query without a tampered player action query. Runs like R7 and R10 show that tampered player action queries can be tolerated without necessarily causing tampered update area queries. Furthermore, even if they lead to some tampered update area queries, data

Table 6.35: Tampered player action and update area retrievals for $r = 9$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
15	0	0	0	0	1	0	1	0	0	0
	0	0	0	0	0	0	0	0	0	1
18	1	0	2	0	1	0	0	1	0	1
	0	0	5	0	0	0	0	0	0	1
19	0	0	2	0	0	0	0	0	0	1
	0	0	5	0	0	0	0	0	0	1
20	0	0	3	0	1	1	0	2	2	1
	0	0	103	0	0	0	0	0	0	1
25	1	0	8	0	3	0	0	4	0	13
	0	0	105	0	0	0	0	0	0	432
30	12	7	11	3	3	1	2	2	1	18
	85	0	109	0	0	0	251	1	0	435
35	18	9	17	2	13	6	6	6	3	49
	165	0	224	0	459	0	300	1	0	482
40	37	14	46	9	25	8	29	14	6	66
	579	309	652	227	492	7	861	281	0	854
50	60	113	66	37	61	36	78	82	28	87
	710	1320	920	965	1005	626	1321	1212	603	1472
75	450	496	250	473	421	204	622	465	298	942
	3269	3833	2983	3655	2932	2601	3522	3508	3161	3742

Table 6.36: Tampered player action and update area retrievals for $r = 4$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
1	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	56	0	0	0	0
4	0	0	1	0	0	0	0	0	0	0
	0	0	0	0	0	57	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	56	0	0	0	0
6	0	0	0	0	0	2	0	0	0	0
	0	0	0	0	0	264	0	0	0	0
7	0	1	0	3	0	2	0	2	0	0
	0	0	0	0	0	269	0	198	0	0
8	4	0	0	0	0	0	0	1	0	38
	24	2	0	0	0	266	0	274	0	386
9	1	2	1	0	0	0	0	4	0	37
	0	51	278	0	0	263	168	282	0	399
10	2	0	2	3	1	1	2	10	0	37
	0	0	292	171	192	268	341	430	166	393

Table 6.37: Tampered player action and update area retrievals for $r = 16$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
35	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0	0	0
37	0	0	1	0	0	0	0	1	0	1
	0	0	6	0	0	0	0	0	0	0
38	0	0	1	0	0	0	1	2	0	0
	0	0	6	0	0	0	0	0	0	0
39	0	0	6	0	0	0	0	2	0	1
	0	0	333	0	0	0	0	0	0	0
40	0	1	2	0	0	0	0	2	0	0
	0	0	334	0	0	0	0	0	0	0
45	0	0	2	0	0	1	2	3	1	0
	0	0	335	0	0	0	0	135	0	1
49	0	0	5	4	3	1	3	13	0	0
	0	0	340	11	0	258	2	355	0	1
60	4	1	10	8	4	8	8	39	8	4
	5	0	459	658	160	616	17	808	349	11
75	17	29	46	64	47	29	33	113	75	31
	330	871	873	1262	513	1006	794	1514	1445	926
100	143	113	490	362	208	280	271	476	505	162
	2592	2578	3427	3326	2507	3320	3083	3517	3599	2607

is not necessarily tampered successfully as can be seen in run R3 with 37 and 38 malicious nodes, not creating tampered data as shown in table 6.31.

Finally, we also counted the number of AOI retrievals returning tampered results as shown in table 6.38 for $r = 9$. If data was tampered as shown in table 6.29, there will definitely be AOI queries returning tampered results if player characters are in the area of tampered data. There are some runs returning tampered AOIs without data in the storage being tampered e.g. run R2 with 40 malicious nodes, meaning malicious nodes were able to intercept the query and answer it instead. However, compared to the around 6000 total AOI retrievals (see table 6.17, the vast majority of AOI retrievals returned correct results if data was not tampered.

This situation was very similar with $r = 4$ as shown in table 6.39. There, run R6 with four and five malicious nodes even had no tampered AOI retrievals although

Table 6.38: Tampered AOI retrievals for $r = 9$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
15	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0
20	0	0	13	0	0	0	0	0	0	0
25	0	0	13	0	0	0	0	0	0	53
30	4	0	14	0	0	0	13	1	0	59
35	2	0	16	2	58	0	15	1	0	44
40	59	40	106	11	63	1	104	32	0	109
50	74	178	146	92	117	75	117	121	62	204
75	482	753	407	630	530	387	620	610	496	778

data was tampered. Apparently, no player characters were in the tampered area. Only very few AOI retrieves got tampered when there was no tampered data like in run R1 with nine and ten malicious nodes.

With $r = 16$ shown in table 6.40, there was no single run where tampered AOI results were returned without any honest node containing tampered data as shown in table 6.31. Furthermore, only run R7 with 60 malicious nodes returned tampered AOI results without data being tampered successfully. Thus, with higher replication factors, AOI retrieves are very resilient although their path length is larger than the path length of update area retrievals.

6.4.5 Discussion

Drawing final conclusions about the long-term reliability of our storage is a little problematic since our test runs have been very short since performing tests was so time-consuming. We opted to evaluate the storage in a larger variety of different scenarios instead of just performing a few long runs with churn being the only source of variation.

Considering our test scenarios with $r = 4$ and $r = 9$, even $r = 4$ was sufficient to not lose data and to update the state of the world correctly over the course of the test runs. As no more than two replication zones had nodes joining or failing in the same are in the same update interval, the majority voting caused all replicas to regain a consistent and correct state at the end of the run. Over longer durations,

Table 6.39: Tampered AOI retrievals for $r = 4$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	32	0	0	0	0
7	0	0	0	1	0	34	0	51	0	0
8	10	0	0	1	0	38	0	103	0	71
9	1	18	81	1	0	35	46	81	0	70
10	2	0	76	22	45	34	102	177	19	69

Table 6.40: Tampered AOI retrievals for $r = 16$

m	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
35	0	0	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0	0	0
39	0	0	9	0	0	0	0	0	0	0
40	0	0	14	0	0	0	0	0	0	0
45	0	0	2	0	0	0	0	2	0	0
49	0	0	11	0	0	5	0	6	0	0
60	0	0	27	35	8	29	4	57	15	0
75	15	63	32	62	24	43	37	157	120	79
100	232	202	455	338	274	261	351	481	390	215

more than two zones experiencing join or leave will probably happen so using higher replication factors like $r = 9$ would probably be advisable.

With $r = 16$, and smaller node numbers, the storage is not reliable in all scenarios any more and a weakness becomes apparent: high bandwidth consumption. Bandwidth requirements become so high that some nodes are not able to serve queries and complete their queries, leading to incorrect and inconsistent updates in different zones and even to complete loss of objects.

Considering the average bandwidth consumptions, the storage scales pretty well with increasing node numbers as the bandwidth consumption actually decreases if the world stays the same. However, this also means bandwidth consumption increases with decreasing node numbers and it also increases with higher replication factors as both lead to larger Voronoi cells per node. Each node, has to retrieve larger update areas receiving more data in return and it also receives more queries from other nodes it has to serve.

Unfortunately, the distribution of Voronoi cells governing the distribution of traffic is not guaranteed to be even. There are some nodes with much larger areas than the average Voronoi cell size, causing these nodes to have a much higher bandwidth consumption than the rest. Apparently, performing a Voronoi tessellation on randomly picked position does not create an even distribution, even if a rasterization guarantees a minimum distance between positions. The inability to balance load at runtime is a serious drawback of our approach, causing overload on nodes with large Voronoi cells. However, using fixed CA-assigned IDs to distribute data is the fundamental idea to prevent tampering, otherwise an attacker could just direct the entire load to him by supplying abundant resources.

Since load balancing is not possible, every node needs sufficiently high bandwidth to shoulder a much higher than average load, depending on the distances to the nearest nodes currently present. In the smaller scenarios with $r = 16$, even our assumed 100 MBit/s symmetric bandwidth exclusively available for our storage was not adequate in all cases. In practice, with other applications running concurrently generating traffic, it will probably take decades until all nodes of a network can be guaranteed to have available bandwidth in this range. Otherwise, just a few slower nodes with larger areas can make the storage unreliable.

Although decreasing bandwidth consumptions with increasing number of node seems to be perfectly scalable, there are two kinds of limits for scalability. First, the bandwidth only decreases because the world stayed the same. We designed it for 5000 players but had a maximum of 1000 players in the world, because in reality, only

a fraction of players will be online at the same time, too. As player numbers increase, at some point the world has to become larger and more objects have to be added to the world, increasing bandwidth consumption again. In the end, each player expects a certain amount of objects to be available for him to interact with them. The world should not feel empty with players having to compete for interacting with the objects in the world.

Second, as Voronoi cell sizes shrink with increasing node numbers, the requested update area sizes increases in relation leading to update area queries overlapping more nodes. As more nodes are involved in answering the query for one zone, chances a malicious node is part of it and spoils the result increase, making the storage less resilient.

Despite the storage being resilient to a certain number of malicious nodes depending on the replication factor, the level of resilience is actually way below our expectations. We had hoped that the storage is guaranteed to withstand 10% to 20% of nodes being malicious attackers with reasonable replication factors like $r = 9$ or $r = 16$. However, to reach these numbers even higher replication factors would be necessary at the cost of another considerable increase in bandwidth consumption. Assuming sizes of Voronoi cells remain the same, the bandwidth consumption will increase linearly with the replication factor. However, with the same number of nodes increasing r also increases the Voronoi cell of a node by a linear amount. Although the storage scales well with node numbers, this $O(r^2)$ dependency of the bandwidth consumption causes improving resilience to be very expensive bandwidth-wise.

The cause of this low level of resilience is the sensitivity of the state update process and the many ways an attacker has to tamper with it. Correct updating is only possible if a majority of zones answered with a correct result without even one of the nodes being malicious. If a malicious node is in the top-layer of the requesting node's routing table, it can reply with any forged state. If it is just one of multiple answering nodes, it can either tamper with state if it controls areas overlapping in multiple zones or it can choose to let the query in that zone fail to lower votes for correct results.

Afterwards, all queries to retrieve the action of players in the update area have to return a correct result, despite the queries possibly being routed over multiple hops giving an attacker more chances to prevent successful return of data from a zone. Furthermore, malicious nodes can also drop actions to be stored instead of forwarding them. Since the storage also has to compensate occasional node join and

leave in different zones, a one-time bad combination of different factors for malicious nodes to exert influence can lead to an all-time tampering of state.

From all the different factors, the need to retrieve all player actions correctly for a correct state update proved to be the main cause for the sensitivity. However, this also means that malicious nodes could mainly indirectly tamper by dropping actions of honest player. It was not possible for them to modify the world state directly. This also opens up a possible future improvement. If retrieving actions of players is much more sensitive, the replication factor could be increased even more for player actions only – either by mapping actions to more position or by using another second overlay or standard DHT with a much higher replication factor. As actions are much smaller than world state, increasing the replication factor would be possible without a drastic increase in bandwidth consumption.

One issue of practical relevance not appearing in our tests is variation of node numbers over time. We used stable churn rates keeping the numbers about constant. However, in practice the number of players in an MMORPG varies greatly. It is usually highest in the evening and lowest in the night. An attacker not controlling enough nodes to tamper in the evening could just wait for the night to come to increase its relative share to allow tampering. Thus, additional means have to be employed to prevent this. One could design the virtual world in a way so it appeals to players all over the world and the number of player does not vary much. Since a lot of players prefer a world where all people use the same language, it could also be possible to simulate multiple virtual worlds on the same overlay also aiming at keeping the numbers of nodes constant.

Chapter 7

Conclusions

To conclude this thesis, we will first summarize the results we achieved and the lessons learned. Afterwards, we will give an outlook on future directions that could be followed on the path towards real peer-to-peer-based MMVEs.

7.1 Summary

Realizing virtual worlds based on peer-to-peer technology has spawned a lot of interest as it could improve scalability compared to current client/server-based MMVEs and more importantly, it could reduce costs of operating the MMVE. For all virtual worlds, cheating is a serious threat. Most MMVEs being games involving competition among players will lead to some players trying to cheat and break the rules of the virtual world to get an unfair advantage. Players want a fair competition. If cheating is possible, they will not play the game and look for a better one.

There are two types of rules defined by the operator, we called them codified rules and uncoded rules. Codified rules are in-world rules implemented in executable code specifying how the world looks like, what actions can be performed by players, and how the world evolves according to player actions. Through uncoded rules, the operator imposes additional restrictions on player behaviour, e.g. by forbidding game automation or collusion. As uncoded rules are concerned with the real world in contrast to codified rules, there is no way to prevent breaking them. Cheaters breaking these rules can only be detected and punished and procedures to do this must be in place in client/server-based as well as peer-to-peer-based MMVEs.

However, peer-to-peer-based MMVEs open up additional ways of breaking codified rules compared to a client/server solution. A central server is the only authority

on the world state, updating it according to actions sent by the players. For every action, it can always decide whether this action is possible according to the rules and apply the correct effects of that action. Furthermore, the server is the only source of information about the world for the clients. For a peer-to-peer-based MMVE to be scalable, the state of the world must be distributed among the nodes. Therefore, malicious node can modify the state they are storing. Furthermore, messages are exchanged in a decentralized fashion and malicious nodes can always try to tamper with messages coming their way. Since the nodes are under physical control of the player, a cheater could modify code and data on the node without other nodes knowing whether he is honest or acts maliciously.

So far, approaches to realize peer-to-peer-based MMVEs have either not considered cheating or they relied on the existence of trusted nodes known to be honest to deal with cheating. We were interested to find out whether dealing with cheating is also possible without trusted honest nodes, just assuming only a certain percentage of node belongs to cheaters. We have developed a hybrid architecture consisting of an overlay allowing local event and state exchange to perform the distributed simulation of the virtual world around the current position of a player and a persistent storage performing a distributed simulation of the whole world while protecting the world state from being tampered with. This way, there is always a reliable distributed authority nodes can use to retrieve the correct state.

Our storage is based on two main ideas. First, to make sure game state only evolves according to codified rules, a correct state should only be calculated based on an old state known to be correct and the actions performed by players. Thus, every node can check whether executing an action is allowed based on the old state and calculate the new state based on the rules. Analysing this update process, we observed the state in an area at a certain time depends on the state in a larger area around this area at an older time. The added radius depends on two factors: the maximum speed at which effects can propagate in the virtual world and the age of the old state, yielding a maximum influence range of objects that can have an influence on the state in the area to update. However, this dependency is the main reason why the local event exchange overlay alone cannot ensure world state to update correctly.

Therefore, our second idea is to replicate the world in different zones, to allow successful retrieving of the old correct state in the presence of malicious nodes. In every zone, an event-based simulation of the world is performed leading to consistent state evolution among all zones in areas not suffering from late messages or tampering

by malicious nodes. Nodes can retrieve the replicas from the zones independently from one another and perform a majority voting to retrieve the correct state. Since malicious nodes cannot influence the areas they are responsible for, there is a high probability for the state to be correct depending on the number of malicious nodes and the chosen replication factor.

In addition to preventing state tampering, timing cheats, inconsistency cheats, and information exposure can be prevented to a certain degree. Since nodes must store their generated actions before a certain time in all replication zones, there is no use for cheaters to delay these messages. However, they can still delay or drop these actions on the local event exchange overlay, effectively causing honest players to see a wrong state until the next lookup of the state using the storage. Since it is impossible to prove a cheater has not sent a message if he claims he did, a detection and punishment procedure could be used. Furthermore, a tit-for-tat strategy blinding or delaying actions to players blinding an honest player could be used in the local event exchange. At the same time, a malicious node sending different actions to different players to cause inconsistencies can be detected and punished quite easily as it sent out the proof itself. Within the storage, a malicious node can always cause inconsistencies by tampering with its replicas and the majority voting will compensate for that with a certain probability. Finally, information exposure is limited as the storage will only give out area information to nodes that are currently in that area or that are responsible for updating that area. This way, a single malicious node can only obtain information beyond its in-world perception range in the area it is actually responsible for.

We have shown the storage to store and update data reliably without the presence of malicious nodes. We have also shown it to scale well with the number of nodes. However, the absolute numbers for bandwidth consumption are very high with high replication factors, way beyond the capacity of current DSL connections. Furthermore, some nodes face a much higher load than others. We cannot allow dynamic load balancing based on available resources, since this could be exploited by an attacker by providing more resources than others. Therefore, every node can face a high load depending on which other nodes are currently present and all nodes' bandwidth capacities must be able to accommodate these high loads. The level of resilience we could achieve with replication factors up to 16 was still quite low. The share of coordinated malicious nodes the storage could tolerate with $r = 16$ was only 4% during short tests. With more diverse scenarios and longer tests runs this number might still be too optimistic. On the other hand, bringing 40 coordinated malicious nodes to attack an overlay of 1000 nodes still requires some effort by an

attacker. This number can be increased by further increasing the replication factor. However, bandwidth consumption grows with the square of the replication factor if the number of nodes stays the same so it does not scale well with the replication factor. In the next section, we will discuss some ideas to reduce the bandwidth consumption to allow the storage to tolerate even more malicious nodes.

The storage is based on a peer-to-peer overlay allowing communication among nodes and also realizing a mapping function to partition the data space into Voronoi cells and assign responsibilities for storing data. Similar to other structured peer-to-peer overlays realizing a distributed hash table, it allows to use the data key as an address to send messages to the responsible node to store and retrieve data. In our overlay, keys are the two-dimensional positions of objects. By connecting nodes of neighbouring Voronoi cells, the overlay creates a Delaunay graph structure allowing executing range queries efficiently. This structure is maintained using a self-stabilizing algorithm.

We map the area of the virtual world to the ID space of the overlay and pick node IDs from that space in a special way to partition the ID space into different zones where nodes in each zone can perform their own simulation of the virtual world. Using three-dimensional routing tables, we realize short routing paths while allowing the retrieval of data from different replication zones on disjoint paths as required by the storage. We have shown the overlay maintenance to work reliably and to scale well with the number of nodes with bandwidth requirements far below the requirements of the storage.

Furthermore, the overlay maintenance is resilient against malicious nodes trying to increase their influence in the overlay beyond their proportional share. This resilient maintenance is necessary for redundant message exchange and voting to contain the influence of cheaters to work. However, malicious nodes still have ways to cause temporary disorder in the overlay. This behaviour can only be detected and nodes should be punished by exclusion to prevent it in the future.

To conclude, we have shown that using redundancy to prevent state tampering in a peer-to-peer-based MMVE of untrusted nodes works in principle. The bandwidth required to realize a high level of resilience is very high. There are still open issues and opportunities for improvements, we will discuss in the next section.

7.2 Outlook

While we have shown that state tampering and related cheats can be prevented with a certain probability, there are other types of cheats and misbehaviour that can only be detected and punished. Some of these can be detected quite easily if the cheater sends out the proof of cheating like he does when he sends out conflicting actions. This proof can be presented to the CA and he can be punished by revoking his certificate. Other misbehaviour like blinding players or causing temporary disorder cannot be proven. If a single node detecting misbehaviour can punish another node by just accusing it, then a cheater could easily accuse and punish honest nodes. If accusations are raised it is only assured that one of the involved parties is a cheater. To deal with these kinds of misbehaviour, detection and punishment strategies still need to be developed.

To improve the excessive bandwidth requirements with high replication factors, several techniques could be employed to reduce the amount of data transferred. As a basic improvement, compressing data with some general purpose compression should help to reduce bandwidth. Furthermore, from one update to the next, only a small fraction of the variables of each object actually change, the position being the most likely one for mobile objects. Therefore, a requesting node could include hashes of the old state of objects into its query. The receiving nodes can check whether this hash actually matches their old version of the data and only return the variables of each object that have changed, while additional objects the requesting node did not store before or non-matching objects are returned completely. This should lead to a drastic reduction in bandwidth consumption. Another way to further decrease bandwidth consumption would be to remove NPCs from areas without player characters and to spawn them in a deterministic way according to the rules if players get closer to an area. However, this would only decrease bandwidth consumption on nodes that are not facing a high load anyway.

To solve the problem of very uneven load distribution, the operator could assign positions dynamically upon node join by giving out certificates with shorter durations that need to be renewed instead of just assigning one static position. He could keep track of the nodes in the network and assign positions in a way that new nodes are placed where nodes currently have large Voronoi cells. While bandwidth requirements for managing joining and leaving nodes will be higher than just creating certificates for new players, it will still be much lower than a server actively involved in simulating the world. Furthermore, the operator needs to supply a bootstrapping

service that malicious nodes cannot tamper with anyway and using a trusted server might be the best solution for this purpose.

There are also options for improving the resilience of the storage without increasing the replication factor. Detection and punishment could also be employed as part of the storage to detect nodes providing false answers and remove them upon detection. If a node intercepting a query replies with a suspiciously big responsibility area, additional queries can be sent out to prove this node reported an incorrect neighbourhood. Furthermore, multipath routing as used by the overlay to fill routing tables could be used to route user messages on alternating paths as well. Due to the high sensitivity of missing player actions leading to incorrect state updates, using higher replication factors for player actions only might also increase resilience without consuming too much bandwidth.

Finally, simulating multiple virtual worlds on the same overlay might help to counter the problem of varying player numbers over the course of the day which could be exploited by cheaters to increase their influence. At the same time, this could also help to balance load and further limit information exposure e.g. by preferring nodes to update areas in worlds their players are not in.

We think it would be worthwhile to explore some of the ideas we mentioned so peer-to-peer-based MMVEs using untrusted nodes only have a chance of becoming reality.

Bibliography

- [1] ABAN, I. B., MEERSCHAERT, M. M., AND PANORSKA, A. K. Parameter Estimation for the Truncated Pareto Distribution. *Journal of the American Statistical Assoc* 101 (2006).
- [2] ALBANO, M., RICCI, L., BALDANZI, M., AND BARAGLIA, R. VoRaQue: Range queries on Voronoi overlays. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on* (July 2008), pp. 495–500.
- [3] ARENANET. Guild Wars. published in the WWW at <http://www.guildwars.com>, April 2005. Last visited in January 2013.
- [4] ARENANET. Guild Wars 2. published in the WWW at <http://www.guildwars2.com>, August 2012. Last visited in January 2013.
- [5] ARTIGAS, M. S., LÓPEZ, P. G., AND GÓMEZ-SKARMETA, A. F. A Novel Methodology for Constructing Secure Multipath Overlays. *IEEE Internet Computing* 9, 6 (2005), 50–57.
- [6] ASSIOTIS, M., AND TZANOV, V. A distributed architecture for MMORPG. In *NETGAMES* (2006), A. D. Cheok and Y. Ishibashi, Eds., ACM, p. 4.
- [7] AURENHAMMER, F. Voronoi Diagrams – A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys* 23, 3 (1991), 345–405. Habilitationsschrift. [Report B 90-09, FU Berlin, Germany, 1990].
- [8] BACKHAUS, H., AND KRAUSE, S. QuON: a quad-tree-based overlay protocol for distributed virtual worlds. *IJAMC* 4, 2 (2010), 126–139.
- [9] BARTLE, R. *Designing Virtual Worlds*. New Riders Games, 2003.
- [10] BARTLE, R. A. Pitfalls of Virtual Property. The Themis Group, 2004.
- [11] BAUGHMAN, N. E., LIBERATORE, M., AND LEVINE, B. N. Cheat-proof payout for centralized and peer-to-peer gaming. *IEEE/ACM Trans. Netw.* 15, 1 (2007), 1–13.

- [12] BAUMGART, I., HEEP, B., AND KRAUSE, S. OverSim: A scalable and flexible overlay framework for simulation and real network applications. In *Peer-to-Peer Computing, 2009. P2P '09. IEEE Ninth International Conference on* (Sept. 2009), pp. 87–88.
- [13] BEAUMONT, O., KERMARREC, A.-M., MARCHAL, L., AND RIVIERE, E. VoroNet: A scalable object network based on Voronoi tessellations. In *IPDPS* (2007), IEEE, pp. 1–10.
- [14] BHARAMBE, A., PANG, J., AND SESHAN, S. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *Proceedings of the 3rd ACM/USENIX Symposium on Network Design and Implementation (NSDI'06)* (San Jose, CA, USA, 2006).
- [15] BHARAMBE, A. R., AGRAWAL, M., AND SESHAN, S. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM* (2004), R. Yavatkar, E. W. Zegura, and J. Rexford, Eds., ACM, pp. 353–366.
- [16] BLIZZARD ENTERTAINMENT INC. World of Warcraft. published in the WWW at <http://www.worldofwarcraft.com>, November 2004. Last visited in January 2013.
- [17] BROOKE, P. J., PAIGE, R. F., CLARK, J. A., AND STEPNEY, S. Playing the Game : cheating, loopholes, and virtual identity. *ACM SIGCAS Computers and Society* 34(2) (Sept. 01 2004), 3.
- [18] BUYUKKAYA, E., AND ABDALLAH, M. Data Management in Voronoi-Based P2P Gaming. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (Jan. 2008), pp. 1050–1053.
- [19] BUYUKKAYA, E., ABDALLAH, M., CAVAGNA, R., AND HU, S.-Y. GROUP: Dual-Overlay State Management for P2P NVE. In *ICPADS '08 Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems* (2008).
- [20] CALVIN, J., DICKEN, A., GAINES, B., METZGER, P., MILLER, D., AND OWEN, D. The SIMNET virtual world architecture. *Proceedings of VRAIS'93* (1993), 450–455.
- [21] CASTRO, M., DRUSCHEL, P., GANESH, A. J., ROWSTRON, A. I. T., AND WALLACH, D. S. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proceedings of the 5th ACM Symposium on Operating System Design*

- and Implementation (OSDI-02)* (New York, Dec. 9–11 2002), Operating Systems Review, ACM Press, pp. 299–314.
- [22] CASTRO, M., DRUSCHEL, P., KERMARREC, A., AND ROWSTRON, A. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications* 20, 8 (October 2002), 1489–1499.
- [23] CASTRONOVA, E. Virtual Worlds: A First-Hand Account of Market and Society on the Cyberian Frontier, Nov. 20 2001.
- [24] CHAN, L., YONG, J., BAI, J., LEONG, B., AND TAN, R. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *NETGAMES* (2007), G. J. Armitage, Ed., ACM, pp. 37–42.
- [25] CHAN, M.-C., HU, S.-Y., AND JIANG, J.-R. An efficient and secure event signature (EASES) protocol for peer-to-peer massively multiplayer on-line games. *Computer Networks (Amsterdam, Netherlands: 1999)* 52, 9 (June 2008), 1838–1845.
- [26] CHANG FENG, W., KAISER, E. C., AND SCHLUESSLER, T. Stealth measurements for cheat detection in on-line games. In *NETGAMES* (2008), M. Claypool, Ed., ACM, pp. 15–20.
- [27] CHEN, B. D., AND MAHESWARAN, M. A Fair Synchronization Protocol with Cheat Proofing for Decentralized Online Multiplayer Games. In *NCA* (2004), IEEE Computer Society, pp. 372–375.
- [28] CHUL KIM, K., YEOM, I., AND LEE, J. HYMS: A hybrid MMOG server architecture. *IEICE Transactions* 87-D, 12 (2004), 2706–2713.
- [29] CIKIC, S., GROTTKE, S., LEHMANN-GRUBE, F., AND SABLATNIG, J. Cheat-prevention and -analysis in online virtual worlds. In *Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop* (ICST, Brussels, Belgium, Belgium, 2008), e-Forensics '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 13:1–13:7.
- [30] CORMAN, A. B., DOUGLAS, S., SCHACHTE, P., AND TEAGUE, V. A Secure Event Agreement (SEA) protocol for peer-to-peer games. In *The First International Conference on Availability, Reliability and Security (ARES)* (2006), IEEE Computer Society, pp. 34–41.

- [31] CRONIN, E., FILSTRUP, B., AND JAMIN, S. Cheat-Proofing Dead Reckoned Multiplayer Games. In *International Conference on Application and Development of Computer Games* (Jan. 2003).
- [32] CRONIN, E., FILSTRUP, B., KURC, A. R., AND JAMIN, S. An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games* (New York, NY, USA, 2002), ACM, pp. 67–73.
- [33] DABEK, F., COX, R., KAASHOEK, M. F., AND MORRIS, R. Vivaldi: a decentralized network coordinate system. In *SIGCOMM* (2004), R. Yavatkar, E. W. Zegura, and J. Rexford, Eds., ACM, pp. 15–26.
- [34] DENAULT, A., AND KIENZLE, J. The perils of using simulations to evaluate Massively Multiplayer Online Game performance. In *SIMUTools* (2010), L. F. Perrone, G. Stea, J. Liu, A. Uhrmacher, and M. Villén-Altamirano, Eds., ICST/ACM, p. 4.
- [35] DIJKSTRA, E. W. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM* 17, 11 (Nov. 1974), 643–644.
- [36] DOUCEUR, J. R. The Sybil Attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, Eds., vol. 2429 of *Lecture Notes in Computer Science*, Springer, pp. 251–260.
- [37] DOUGLAS, S., TANIN, E., HARWOOD, A., AND KARUNASEKERA, S. Enabling Massively Multi-Player Online Gaming Applications on a P2P Architecture. In *In Proceedings of the IEEE International Conference on Information and Automation* (2005), IEEE, pp. 7–12.
- [38] ELECTRONIC ARTS. Ultima Online. published in the WWW at <http://www.uo.com>, Sept. 1997. Last visited in January 2013.
- [39] ELECTRONIC ARTS. Star Wars: The Old Republic. published in the WWW at <http://www.swtor.com>, December 2011. Last visited in January 2013.
- [40] EVEN BALANCE INC. Punkbuster Online Countermeasures. published in the WWW at <http://www.evenbalance.com>, Sept. 2000.
- [41] FERRETTI, S. Cheating detection through game time modeling: A better way to avoid time cheats in P2P MOGs? *Multimedia Tools Appl* 37, 3 (2008), 339–363.

- [42] FERRETTI, S., AND ROCCETTI, M. AC/DC: an Algorithm for Cheating Detection by Cheating. In *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video* (New York, NY, USA, 2006), NOSSDAV '06, ACM, pp. 23:1–23:6.
- [43] FERRETTI, S., ROCCETTI, M., AND ZIONI, R. A Statistical Approach to Cheating Countermeasure in P2P MOGs. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference* (Piscataway, NJ, USA, 2009), CCNC'09, IEEE Press, pp. 1267–1271.
- [44] FRIEDMAN, E. J. The Social Cost of Cheap Pseudonyms. *Journal of Economics and Management Strategy* 10, 2 (2001), 173–199.
- [45] FUJIMOTO, R. M. *Parallel and Distributed Simulation Systems*. Wiley Interscience, New York, Jan. 2000.
- [46] GARCÍA, I., MOLLÁ, R., AND BARELLA, T. GDESK: Game discrete event simulation kernel. In *Journal of WSCG* (Plzen, Czech Republic, Feb. 2004), V. Skala, Ed., vol. 12, UNION Agency - Science Press.
- [47] GARTNER, F. C. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Tech. rep., EPFL Lausanne, Distributed Programming Laboratory, June 30 2004.
- [48] GAUTHIERDICKY, C., ZAPPALA, D., LO, V., AND MARR, J. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video* (New York, NY, USA, 2004), ACM, pp. 134–139.
- [49] GAUTIER, L., AND DIOT, C. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In *ICMCS* (1998), pp. 233–236.
- [50] GHAFFARI, M., HARIRI, B., AND SHIRMOHAMMADI, S. A delaunay triangulation architecture supporting churn and user mobility in MMVEs. In *NOSSDAV* (2009), W. T. Ooi and D. Xu, Eds., ACM, pp. 61–66.
- [51] GHODSI, A., ALIM, L. O., AND HARIDI, S. Symmetric Replication for Structured Peer-to-Peer Systems. In *DBISP2P* (2005), G. Moro, S. Bergamaschi, S. Joseph, J.-H. Morin, and A. M. Ouksel, Eds., vol. 4125 of *Lecture Notes in Computer Science*, Springer, pp. 74–85.

- [52] GILMORE, J., AND ENGELBRECHT, H. Pithos: a state persistency architecture for peer-to-peer massively multiuser virtual environments. In *Haptic Audio Visual Environments and Games (HAVE), 2011 IEEE International Workshop on* (oct. 2011), pp. 1–6.
- [53] GOODMAN, J., AND VERBRUGGE, C. A Peer Auditing Scheme for Cheat Elimination in MMOGs. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games* (New York, NY, USA, 2008), ACM, pp. 9–14.
- [54] GORLATCH, S., MEILAENDER, D., BARTOLOMEUS, S., FUJITA, H., THEURL, T., HOEREN, T., HEGHMANN, M., AND BOERS, K. Cheating Prevention in Virtual Worlds: Software, Economic, and Law Aspects. In *Proceedings of the 2010 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT10* (Amsterdam, The Netherlands, The Netherlands, 2010), IOS Press, pp. 268–289.
- [55] GROTTKE, S., SABLATNIG, J., CHEN, J., SEILER, R., KÖPKE, A., AND WOLISZ, A. Measures for Inconsistency in Distributed Virtual Environments. In *ICPADS* (2008), IEEE, pp. 859–864.
- [56] GROTTKE, S., SABLATNIG, J., KÖPKE, A., CHEN, J., SEILER, R., AND ., A. W. Consistency in Distributed Systems. Tech. rep., TKN Technical Report Series TKN-08-005, Telecommunication Networks Group, Technische Universität Berlin, 2008.
- [57] GUMMADI, P. K., SAROIU, S., AND GRIBBLE, S. D. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Computer Communication Review* 32, 1 (2002), 82.
- [58] HAMPEL, T., BOPP, T., AND HINN, R. A peer-to-peer architecture for massive multiplayer online games. In *NETGAMES* (2006), A. D. Cheok and Y. Ishibashi, Eds., ACM, p. 48.
- [59] HARVESF, C., AND BLOUGH, D. M. The Effect of Replica Placement on Routing Robustness in Distributed Hash Tables. In *Peer-to-Peer Computing* (2006), A. Montresor, A. Wierzbicki, and N. Shahmehri, Eds., IEEE Computer Society, pp. 57–66.
- [60] HERLIHY, M. P., AND WING, J. M. Axioms for concurrent objects. In *POPL '87. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles*

- of programming languages, January 21–23, 1987, Munich, W. Germany (pub-ACM:adr, 1987), ACM, Ed., ACM Press, pp. 13–26.
- [61] HERNANDEZ, J., AND PHILLIPS, I. Weibull mixture model to characterise end-to-end Internet delay at coarse time-scales. *Communications, IEE Proceedings- 153*, 2 (april 2006), 295 – 304.
 - [62] HERRERA, O., AND ZNATI, T. Modeling Churn in P2P Networks. In *Annual Simulation Symposium* (2007), IEEE Computer Society, pp. 33–40.
 - [63] HOGLUND, G., AND MCGRAW, G. *Exploiting Online Games: Cheating Massively Distributed Systems*. Addison-Wesley Longman, 2007.
 - [64] HOLZAPFEL, S., SCHUSTER, S., AND WEIS, T. VoroStore - A Secure and Reliable Data Storage for Peer-to-Peer-Based MMVEs. In *IEEE 11th International Conference on Computer and Information Technology (CIT 2011)* (September 2011).
 - [65] HOLZAPFEL, S., WACKER, A., WEIS, T., AND WANDER, M. An Architecture for Complex Peer-to-Peer Systems. In *Proceedings of the 4th IEEE International Workshop on Digital Entertainment, Networked Virtual Environments, and Creative Technology, DENVECT 2012, held in conjunction with the IEEE Consumer Communications and Networking Conference, CCNC 2012* (Las Vegas, Nevada, USA, January 2012).
 - [66] HOLZAPFEL, S., WANDER, M., WACKER, A., AND WEIS, T. SYNI - TCP Hole Punching Based on SYN Injection. In *Proceedings of the The 10th IEEE International Symposium on Network Computing and Applications (IEEE NCA11)* (Cambridge, MA, USA, August 2011).
 - [67] HU, S.-Y., CHANG, S.-C., AND JIANG, J.-R. Voronoi State Management for Peer-to-Peer Massively Multiplayer Online Games. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE* (Jan. 2008), pp. 1134–1138.
 - [68] HU, S.-Y., CHEN, J.-F., AND CHEN, T.-H. VON: a scalable peer-to-peer network for virtual environments. *Network, IEEE 20*, 4 (July 2006), 22 –31.
 - [69] HU, S.-Y., AND LIAO, G.-M. Scalable peer-to-peer networked virtual environment. In *NETGAMES* (2004), W. chang Feng, Ed., ACM, pp. 129–133.
 - [70] IIMURA, T., HAZEYAMA, H., AND KADOBAYASHI, Y. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NETGAMES* (2004), W. chang Feng, Ed., ACM, pp. 116–120.

- [71] JACOB, R., RITSCHER, S., SCHEIDELER, C., AND SCHMID, S. A Self-stabilizing and Local Delaunay Graph Construction. In *ISAAC* (2009), Y. Dong, D.-Z. Du, and O. H. Ibarra, Eds., vol. 5878 of *Lecture Notes in Computer Science*, Springer, pp. 771–780.
- [72] JARDINE, J., AND ZAPPALA, D. A hybrid architecture for massively multiplayer online games. In *NETGAMES* (2008), M. Claypool, Ed., ACM, pp. 60–65.
- [73] JEFFERSON, D. R. Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 3 (July 1985), 404–425.
- [74] JIANG, J.-R., HUANG, Y.-L., AND HU, S.-Y. Scalable AOI-Cast for Peer-to-Peer Networked Virtual Environments. *Distributed Computing Systems Workshops, International Conference on Distributed Computing Systems 0* (2008), 447–452.
- [75] KABUS, P., AND BUCHMANN, A. P. Design of a Cheat-Resistant P2P Online Gaming System. In *DIMEA* (2007), K. K. W. Wong, L. C. C. Fung, and P. Cole, Eds., vol. 274 of *ACM International Conference Proceeding Series*, ACM, pp. 113–120.
- [76] KABUS, P., TERPSTRA, W. W., CILIA, M., AND BUCHMANN, A. P. Addressing cheating in distributed MMOGs. In *NETGAMES* (2005), ACM, pp. 1–6.
- [77] KAWAHARA, Y., AOYAMA, T., AND MORIKAWA, H. A Peer-to-Peer Message Exchange Scheme for Large-Scale Networked Virtual Environments. *Telecommunication Systems* 25, 3-4 (2004), 353–370.
- [78] KELLER, J., AND SIMON, G. Solipsis: A Massively Multi-Participant Virtual World. In *PDPTA* (2003), H. R. Arabnia and Y. Mun, Eds., CSREA Press, pp. 262–268.
- [79] KNIESBURGES, S., KOUTSOPOULOS, A., AND SCHEIDELER, C. Re-Chord: a self-stabilizing chord overlay network. In *SPAA* (2011), R. Rajaraman and F. M. auf der Heide, Eds., ACM, pp. 235–244.
- [80] KNUTSSON, B., LU, H., XU, W., AND HOPKINS, B. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies* (March 2004), vol. 1, pp. –107.

- [81] KRANAKIS, E., SINGH, H., AND URRUTIA, J. Compass Routing on Geometric Networks. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG-99)* (Aug. 15–18 1999), pp. 51–54.
- [82] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS* (November 2000), ACM.
- [83] LAURENS, P., PAIGE, R. F., BROOKE, P. J., AND CHIVERS, H. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems* (Washington, DC, USA, 2007), ICECCS '07, IEEE Computer Society, pp. 97–106.
- [84] LAZOWSKA, E. D., ZAHORJAN, J., GRAHAM, G. S., AND SECIK, K. C. *Quantitative System Performance*. Prentice Hall, Inc., 1984.
- [85] LI, K., DING, S., MCCREARY, D., AND WEBB, S. Analysis of state exposure control to prevent cheating in online games. In *NOSSDAV* (2004), V. N. Padmanabhan and C. J. Sreenan, Eds., ACM, pp. 140–145.
- [86] LIANG, H., TAY, I., NEO, M. F., OOI, W. T., AND MOTANI, M. Avatar Mobility in Networked Virtual Environments: Measurements, Analysis, and Implications. *CoRR abs/0807.2328* (2008).
- [87] LIEBEHERR, J., NAHAS, M., AND SI, W. Application-layer multicast with Delaunay triangulations. *IEEE Journal on Selected Areas in Communications* 20 (2001), 1472–1488.
- [88] MAROUANI, H., AND DAGENAIS, M. R. Internal Clock Drift Estimation in Computer Clusters. *Journal Comp. Netw. and Communic 2008* (2008).
- [89] MAUVE, M., VOGEL, J., HILT, V., AND EFFELSBERG, W. Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia* 6 (2004), 47–57.
- [90] MILLS, D. L. *Computer Network Time Synchronization: the Network Time Protocol on Earth and in Space*. CRC Press, 2011.
- [91] MISRA, J. Distributed discrete event simulation. *ACM Comput. Surveys* 18 (1986), 39–65.

-
- [92] MMODATA.NET. MMOData Charts v3.8. published in the WWW at <http://www.mmodata.net>, Mar. 2012. Last visited in January 2013.
 - [93] MÖNCH, C., GRIMEN, G., AND MIDTSTRAUM, R. Protecting online games against cheating. In *NETGAMES* (2006), A. D. Cheok and Y. Ishibashi, Eds., ACM, p. 20.
 - [94] MORILLO, P., MONCHO, W., ORDUÑA, J. M., AND DUATO, J. Providing Full Awareness to Distributed Virtual Environments Based on Peer-to-Peer Architectures. In *CGI* (2006), T. Nishita, Q. Peng, and H.-P. Seidel, Eds., vol. 4035 of *Lecture Notes in Computer Science*, Springer, pp. 336–347.
 - [95] MULLIGAN, J., AND PATROVSKY, B. *Developing Online Games. An Insiders Guide*. New Riders, 2003.
 - [96] PHAM, A. Star Wars: The Old Republic – the costliest game of all time? Los Angeles Times Online, 20. Jan. 2012.
 - [97] PITTMAN, D., AND GAUTHIERDICKY, C. A measurement study of virtual populations in massively multiplayer online games. In *NETGAMES* (2007), G. J. Armitage, Ed., ACM, pp. 25–30.
 - [98] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-Addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (San Diego, CA, USA, 2001), ACM Press, pp. 161–172.
 - [99] RICCI, L., AND GENOVALI, L. State management in Distributed Virtual Environments: A Voronoi base approach. In *ICUMT* (2010), IEEE, pp. 881–887.
 - [100] RICCI, L., AND SALVADORI, A. Nomad: Virtual Environments on P2P Voronoi Overlays. In *OTM Workshops (2)* (2007), R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 4806 of *Lecture Notes in Computer Science*, Springer, pp. 911–920.
 - [101] RIECHE, S., WEHRLE, K., FOUQUET, M., NIEDERMAYER, H., PETRAK, L., AND CARLE, G. Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games. In *4th Annual IEEE Consumer Communications and Networking Conference (CCNC 2007)* (Las Vegas, Jan. 2007), IEEE, pp. 763–767.

- [102] RILEY, G. F., AND HENDERSON, T. R. The ns-3 Network Simulator. In *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 15–34.
- [103] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science 2218* (2001), 329–350.
- [104] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM SOSP* (Lake Louise, Alberta, Canada, October 2001).
- [105] SCHMIEG, A., STIELER, M., JECKEL, S., KABUS, P., KEMME, B., AND BUCHMANN, A. P. pSense - Maintaining a Dynamic Localized Peer-to-Peer Structure for Position Based Multicast in Games. In *Peer-to-Peer Computing* (2008), K. Wehrle, W. Kellerer, S. K. Singhal, and R. Steinmetz, Eds., IEEE Computer Society, pp. 247–256.
- [106] SCHUSTER, S., WACKER, A., AND WEIS, T. Fighting Cheating in P2P-based MMVEs with Disjoint Path Routing. *Electronic Communications of the EASST 17* (2009).
- [107] SCHUSTER, S., AND WEIS, T. Enforcing Game Rules in Untrusted P2P-based MMVEs. In *Distributed Simulation & Online Gaming (DISIO)* (2011).
- [108] SEEGER, C., KEMME, B., KABUS, P., AND BUCHMANN, A. P. Area-based gossip multicast. In *NETGAMES* (2008), M. Claypool, Ed., ACM, pp. 40–45.
- [109] SINGH, A., NGAN, T.-W., DRUSCHEL, P., AND WALLACH, D. S. Eclipse Attacks on Overlay Networks: Threats and Defenses. In *INFOCOM* (2006), IEEE.
- [110] SINGHAL, S., AND CHERITON, D. Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. *Presence 4*, 2 (1995), 169–194.
- [111] SIT, E., AND MORRIS, R. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *IPTPS* (2002), P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, Eds., vol. 2429 of *Lecture Notes in Computer Science*, Springer, pp. 261–269.
- [112] SMED, J., AND HAKONEN, H. Preventing Look-Ahead Cheating with Active Objects. In *Computers and Games* (2004), H. J. van den Herik, Y. Björnsson,

- and N. S. Netanyahu, Eds., vol. 3846 of *Lecture Notes in Computer Science*, Springer, pp. 301–315.
- [113] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference* (San Diego, California, United States, 2001), ACM, pp. 149–160.
- [114] STUTZBACH, D., AND REJAIE, R. Understanding churn in peer-to-peer networks. In *Internet Measurement Conference* (2006), J. M. Almeida, V. A. F. Almeida, and P. Barford, Eds., ACM, pp. 189–202.
- [115] SUPERDATA RESEARCH INC. MMO games market analysis 2012. summary available in the WWW at <http://www.superdataresearch.com>, July 2012. Last visited in January 2013.
- [116] TURBINE INC. Lord of the Rings Online. published in the WWW at <http://www.lotro.com>, April 2007. Last visited in January 2013.
- [117] VARGA, A. OMNeT++. In *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 35–59.
- [118] VARVELLO, M., PICCONI, F., DIOT, C., AND BIRSACK, E. W. Is there life in Second Life? In *CoNEXT* (2008), A. Azcorra, G. de Veciana, K. W. Ross, and L. Tassiulas, Eds., ACM, p. 1.
- [119] VERANT INTERACTIVE. Everquest. published in the WWW at <http://www.everquest.com>, Mar. 1999. Last visited in January 2013.
- [120] VOGELS, W. Eventually Consistent. *Queue* 6, 6 (Oct. 2008), 14–19.
- [121] WACKER, A., SCHIELE, G., SCHUSTER, S., AND WEIS, T. Towards an Authentication Service for Peer-to-Peer based Massively Multiuser Virtual Environments. *Int. J. Advanced Media and Communications* (2008).
- [122] WACKER, A., SCHIELE, G., SCHUSTER, S., AND WEIS, T. Towards an Authentication Service for Peer-to-Peer based Massively Multiuser Virtual Environments. In *Proceedings of the The 1st International Workshop on Massively Multiuser Virtual Environments (MMVE08)* (Reno, Nevada, USA, Mar. 2008), IEEE.
- [123] WEBB, S., SOH, S., AND LAU, W. RACS: A Referee Anti-Cheat Scheme for P2P Gaming. In *NOSSDAV'07: 17th International workshop on Network and Operating Systems Support for Digital Audio & Video* (2007), pp. 37–42.

- [124] WEBB, S. D., AND SOH, S. Cheating in networked computer games: a review. In *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts* (New York, NY, USA, 2007), ACM, pp. 105–112.
- [125] WEIS, T., WACKER, A., SCHUSTER, S., AND HOLZAPFEL, S. Towards Logical Clocks in P2P-based MMVEs. *Electronic Communications of the EASST 17* (2009).
- [126] YAN, J. J., AND CHOI, H.-J. Security issues in online games. *The Electronic Library 20*, 2 (2002), 125–133.
- [127] YAN, J. J., AND RANDELL, B. A systematic classification of cheating in online games. In *NETGAMES* (2005), ACM, pp. 1–9.
- [128] YEE, N. Motivations for Play in Online Games. *Cyberpsy, Behavior, and Soc. Networking 9*, 6 (2006), 772–775.
- [129] YU, A. P., AND VUONG, S. T. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV* (2005), W. chi Feng and K. Mayer-Patel, Eds., ACM, pp. 99–104.
- [130] ZHANG, K., KEMME, B., AND DENAULT, A. Persistence in Massively Multiplayer Online Games. In *NETGAMES* (2008), M. Claypool, Ed., ACM, pp. 53–58.
- [131] ZHANG, W., AND HE, J. Modeling End-to-End Delay Using Pareto Distribution. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on* (july 2007), p. 21.

List of Acronyms

2D	two-dimensional
3D	three-dimensional
ADSL	asymmetric digital subscriber line
AI	artificial intelligence
AOI	area of interest
CA	certification authority
DHT	distributed hash table
DSL	digital subscriber line
DVE	distributed virtual environment
FIFO	first in first out
FPS	first-person shooter
GUID	Globally Unique Identifier
IAD	inter-arrival distribution
ID	identifier
IP	Internet Protocol
MMVE	massively multiuser virtual environment
MMOG	massively multiplayer online game
MMOG	massively multiplayer online role-playing game
MTU	maximum transfer unit
MUD	multi-user dungeon
NAT	Network Address Translation
NVE	networked virtual environment
NPC	non-player character
NTP	Network Time Protocol
PVP	player-versus-player
RAM	random-access memory
SLD	session length distribution
TCP	Transmission Control Protocol
XNA	XNA's Not Acronymed

List of Publications

1. HOLZAPFEL, S., SCHUSTER, S., AND WEIS, T. VoroStore - A Secure and Reliable Data Storage for Peer-to-Peer-Based MMVEs. In *IEEE 11th International Conference on Computer and Information Technology (CIT 2011)* (September 2011).
2. SCHUSTER, S., AND WEIS, T. Enforcing Game Rules in Untrusted P2P-based MMVEs. In *Distributed Simulation & Online Gaming (DISIO)* (2011).
3. SCHUSTER, S., WACKER, A., AND WEIS, T. Fighting Cheating in P2P-based MMVEs with Disjoint Path Routing. *Electronic Communications of the EASST 17* (2009).
4. WACKER, A., SCHIELE, G., SCHUSTER, S., AND WEIS, T. Towards an Authentication Service for Peer-to-Peer based Massively Multiuser Virtual Environments. *Int. J. Advanced Media and Communications* (2008).
5. WEIS, T., WACKER, A., SCHUSTER, S., AND HOLZAPFEL, S. Towards Logical Clocks in P2P-based MMVEs. *Electronic Communications of the EASST 17* (2009).
6. WACKER, A., SCHIELE, G., SCHUSTER, S., AND WEIS, T. Towards an Authentication Service for Peer-to-Peer based Massively Multiuser Virtual Environments. In *Proceedings of the The 1st International Workshop on Massively Multiuser Virtual Environments (MMVE08)* (Reno, Nevada, USA, Mar. 2008), IEEE.
7. SCHUSTER, S., AND BRINKSCHULTE, U. Model-driven development of ubiquitous applications for sensor-actuator-networks with abstract state machines. In *SEUS* (2007), R. Obermaisser, Y. Nah, P. Puschner, and F.-J. Rammig, Eds., vol. 4761 of *Lecture Notes in Computer Science*, Springer, pp. 527–536.

List of Figures

2.1	Representation of the virtual world in World of Warcraft as seen by the player	8
4.1	Voronoi tessellation	49
4.2	Delaunay graph	51
4.3	Neighbourhood routing in the Delaunay structure	52
4.4	Pastry routing table, $b=4$, $l=6$	53
4.5	Three layers of a routing table with $b_x = 4$ and $b_y = 4$	54
4.6	Message routing using routing tables	56
4.7	Sending a NetworkView before disconnecting an ex-neighbour	59
4.8	Node b detects failure of node c later than node a and node d	63
4.9	Neighbours of failed node d establish new connections	64
4.10	Eclipsing a joining node	66
4.11	Eclipsing is not possible	66
4.12	Corridor of nodes contacted by joining node a with bootstrapper b . .	68
4.13	JoinRequest path from bootstrapper b to nodes involved in the join of a	69
4.14	Node a with target positions and responsible nodes	72
4.15	Symmetry of Routing Table Relationships	73
4.16	Different receivers for same destination with zone-constrained routing	74
4.17	Closest node b in constrained zone is not a neighbour of closest node a	75
4.18	Only keys from the marked sub-areas are generated	76
4.19	Key space sub-areas for $b_x = 4$ and $b_y = 4$	77
4.20	Node a requests entry $e_{1,0,0}$ with bootstrappers b, c, d, e	80
4.21	Node a requests entry $e_{0,0,0}$ with bootstrappers b, c, d, e but y does not put a into its routing table	83
4.22	Node y passively updates its routing table after joining node a requests entry $e_{0,0,0}$	84
4.23	Node b detects failure of c last and a does send a NetworkView to b after becoming a neighbour of b	90

4.24	Multicast tree for a circular area starting at root node a	93
4.25	Join of node j during multicast leads to message duplication on node f	95
4.26	Node c cannot determine whether parent d of e overlaps and forwards	96
5.1	Mappings s and t map object o to its replica IDs $o_{0,0} \dots o_{1,1}$	104
5.2	Node n retrieving all replicas of an object o creates disjoint paths . .	105
5.3	Player p outside of A at time t_0 moves to arrive inside A at t_1	108
5.4	Interaction chain with transitive effect from player p_1 to p	110
5.5	Circle A^c covering A with increased circle A^{c+d}	113
5.6	A malicious node can only tamper in the sub-area T of its responsibility area due to overlap with honest neighbours	115
5.7	Timing of retrieve operations in the update process	116
5.8	Node a receives PushData messages from its neighbour upon joining .	120
5.9	Failure of neighbour c does not add responsibility area to a	124
5.10	Node a requests its own area overlapping the areas of malicious nodes b and c in other zones	127
5.11	Node a is eclipsed initially but retrieves its true neighbourhood after querying its area A	133
5.12	Nodes only distributed in one dimension lead to every Voronoi cell in one zone overlapping every Voronoi cell in the other zone	137
6.1	NTP-inspired clock resynchronization of drifting clocks	149
6.2	Number of nodes in the network with $E(SLD) = 2h$ and $E(IAD) = 25s$	154
6.3	Number of nodes in the network with $E(SLD) = 2h$ and $E(IAD) = 6s$	158
6.4	Number of fourth layer entries in scenarios IAD6 and IAD25 with $r = 9, r = 16$	167
6.5	Global link layer traffic depending on the number of started nodes . .	171
6.6	Average bandwidth per node depending on the number of started nodes	171
6.7	Frequency distribution of message types for scenario IAD6, $r = 4$. .	173
6.8	Frequency distribution of message types for scenario IAD6, $r = 9$. .	174
6.9	Frequency distribution of message types for scenario IAD6, $r = 16$. .	176
6.10	Global link layer traffic depending on the number of initially started nodes	197
6.11	Average bandwidth per node depending on the number of initially started nodes	198
6.12	Average and maximum added delay due to channel saturation	199
6.13	Malicious nodes overlapping a requested area tamper with data in area T to obtain three votes	213

Listings

4.1	NetworkView processing	60
4.2	Connection state change	61

List of Tables

6.1	Number of started nodes in each run	157
6.2	Number of running nodes at the end of each run	158
6.3	Neighbourhood Sizes Evaluation $r = 4$	159
6.4	Neighbourhood Sizes Evaluation $r = 9$	160
6.5	Neighbourhood Sizes Evaluation $r = 16$	160
6.6	Neighbourhood Sizes Evaluation of Random Delaunay Graphs	161
6.7	Filling degree of the routing table for $r = 4$	162
6.8	Filling degree of the routing table for $r = 9$	164
6.9	Filling degree of the routing table for $r = 16$	166
6.10	Symmetric entries and entries closest to target position averaged over ten test runs	169
6.11	Share of malicious nodes in routing tables	179
6.12	Share of malicious nodes in top layer of routing tables	181
6.13	Average numbers for data stored at the end of all test scenarios . . .	188
6.14	Number of lost objects with $r = 16$	189
6.15	Number of objects missing replicas in at least one of $r = 16$ replication zones	189
6.16	Number of nodes with an incorrect state	190
6.17	Average numbers of queries after update process start	192
6.18	Share of incorrect queries in runs with $r = 16$	193
6.19	Average of query duration medians in ms	195
6.20	Average values for bandwidth distribution characteristics	198
6.21	Average values of the distribution characteristics of the amount of data stored per node	201
6.22	Correlation coefficients of distribution characteristics	202
6.23	Average values for Voronoi cell size distribution characteristics	203
6.24	Average values for maintenance area size distribution characteristics .	203
6.25	Average values for update area retrieval data size distribution char- acteristics	205
6.26	Average values for update area size distribution characteristics	206

6.27	Number of messages per query result	207
6.28	Average hop counts per query type	209
6.29	Honest nodes with tampered data and data with tampered majority for $r = 9$	215
6.30	Honest nodes with tampered data and data with tampered majority for $r = 4$	217
6.31	Honest nodes with tampered data and data with tampered majority for $r = 16$	218
6.32	Share of queries malicious nodes tried to tamper with for $r = 9$	219
6.33	Share of queries malicious nodes tried to tamper with for $r = 4$	220
6.34	Share of queries malicious nodes tried to tamper with for $r = 16$	220
6.35	Tampered player action and update area retrievals for $r = 9$	222
6.36	Tampered player action and update area retrievals for $r = 4$	223
6.37	Tampered player action and update area retrievals for $r = 16$	224
6.38	Tampered AOI retrievals for $r = 9$	225
6.39	Tampered AOI retrievals for $r = 4$	226
6.40	Tampered AOI retrievals for $r = 16$	226